UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Damjan Murn

Analiza pokritosti kode s testi z uporabo mutacijskega operatorja BSR

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2016

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Damjan Murn

Analiza pokritosti kode s testi z uporabo mutacijskega operatorja BSR

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM
RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: viš. pred. dr. Igor Rožanc

Ljubljana, 2016

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pokritost kode s testi je osnovna mera za določanje kakovosti izvedenega testiranja. Običajen pristop z označevanjem obiskanih delov kode med testiranjem uporablja orodje EMMA, ki pa ima tudi nekaj slabosti.

V diplomskem delu zasnujte in realizirajte alternativno rešitev po principu mutacijskega testiranja z uporabo mutacijskega operatorja BSR. Rešitev izvedite kot orodje, ki čim natančneje izdela poročila o pokritosti kode po zgledu orodja EMMA. Poudarek naj bo na uporabnosti in učinkovitosti orodja. Na koncu primerjajte vaše orodje z orodjem EMMA po funkcionalnosti, pravilnosti rezultatov in porabi časa.

Zahvaljujem se mentorju viš. pred. dr. Igorju Rožancu za nasvete in pomoč pri izdelavi diplomskega dela.

Kazalo

Kazalo slik

Kazalo tabel

Seznam uporabljenih kratic

kratica	angleško	slovensko
AST	Abstract Syntax Tree	abstraktno sintaksno drevo
BSR	Bomb Statement Replacement	zamenjava stavka z bombo
HTML	HyperText Markup Language	jezik za označevanje nadbesedila
JAR	Java Archive	javanska arhivska datoteka
MOJO	Maven plain Old Java Object	Maven-ov enostaven javanski objekt
POM	Project Object Model	projektno objektni model

Povzetek

Pokritost programske kode s testi je mera, ki pove, kolikšen delež izvorne kode izbranega programa oziroma poti skozi program je bilo testiranih pri izvajanju testne zbirke. V diplomskem delu smo razvili javansko orodje za analizo pokritosti kode s principi mutacijskega testiranja. Mutacije so majhne modifikacije izvorne kode, ki jih namerno povzročamo v testirani programski kodi z namenom ugotavljanja kvalitete obstoječih testov ter njihove izboljšave. Uporabili smo mutacijski operator BSR ter v javansko izvorno kodo sistematično vstavljali tako imenovane bombe, javanske stavke, ki pri izvajanju povzročijo napako. Na mutirani programski kodi smo poganjali testno zbirko in spremljali, ali se testi uspešno izvedejo. Na osnovi tega smo sklepali na pokritost mesta v kodi, kamor je bila postavljena bomba. Pri tem smo veliko pozornosti posvetili optimizaciji algoritma, ker je mutacijsko testiranje že v osnovi zelo časovno zahteven proces.

Aplikacijo smo razvili kot vtičnik za Maven, orodje za upravljanje javanskih projektov, kar omogoča enostavno uporabo na poljubnem projektu. Za cilj smo si postavili razviti orodje, ki je po funkcionalnosti in načinu merjenja pokritosti podobno orodju EMMA, enemu od javanskih orodij za merjenje pokritosti. Pri testiranju aplikacije na nekaj realnih projektih se je pokazalo, da so rezultati precej natančni in primerljivi z orodjem EMMA.

Ključne besede: pokritost kode s testi, mutacijsko testiranje, mutacijska analiza, mutacijski operator, BSR

Abstract

Code coverage is a measure used to describe the degree to which the source code of a program or paths through the program has been tested by a particular test suite. In this thesis we have designed and implemented a Java tool for code coverage analysis which is based on principles of mutation testing. Mutations are small modifications of source code which are deliberately seeded into the source code under testing in order to evaluate the quality of existing test suite and improve it. We used the mutation operator BSR to systematically seed so-called bombs into Java source code. In mutation testing a bomb is a statement which causes an exception when executed. After that we ran the test suite against the mutated code and checked whether it finished successfully. Based on the test suite results it is possible to assess the code coverage at the position where the bomb was placed. We put a lot of effort in algorithm optimization because mutation testing is known to be extremely time-consuming process.

We designed the application as a plugin for Maven, software project management and comprehension tool, which makes it simple to use on any Maven based Java project. Our goal was to develop a tool similar to EMMA by functionality and code coverage measurement methodology. By running the code coverage analysis on a few real-world open source projects, the application proved to be pretty accurate and comparable to EMMA.

Keywords: test coverage, code coverage, mutation testing, mutation analysis, mutation operator, BSR

Poglavje 1 Uvod

Pokritost programske kode s testi je mera, ki jo uporabljamo pri testiranju programske opreme in pove, kolikšen delež izvorne kode izbranega programa oziroma poti skozi program je bilo testiranih pri izvajanju določene zbirke testov. Gre torej za indikator kakovosti in učinkovitosti testov, s katerimi preverjamo pravilnost delovanja programske opreme. Čim večja je pokritost kode, tem večji delež izvorne kode obravnavanega programa oziroma poti skozi program se je med izvajanjem testov izvedlo in tem manjša je možnost, da program vsebuje skrite napake.

Za merjenje pokritosti kode obstajajo različna orodja, ki spremljajo obravnavano programsko kodo med izvajanjem zbirke testov in pri tem beležijo, kateri deli so se dejansko izvedli. V diplomskem delu smo si postavili za cilj razviti tako orodje, ki bi za razliko od ostalih orodij delovalo po principu mutacijskega testiranja in sicer z uporabo mutacijskega operatorja BSR. V programsko kodo sistematično vstavljamo tako imenovane bombe, javanske stavke, ki pri izvajanju povzročijo izjemo. Če pri izvajanju testne zbirke na mutirani programski kodi pride do izjeme, lahko iz tega sklepamo, da je tisti del kode pokrit. In obratno, če se testna zbirka izvede uspešno kljub vstavljeni bombi, lahko sklepamo, da se tisti del ni izvedel in torej ni pokrit. Zanimalo nas je, kako uporabno bi bilo orodje, ki bi delovalo po tem principu, kakšna bi bila časovna zahtevnost in kako natančno bi bilo v primerjavi z drugimi orodji za merjenje pokritosti kode.

Pri razvoju tega orodja smo si postavili naslednje cilje:

- orodje naj bo čim bolj enostavno za uporabo
- pokritost kode naj meri na podoben način kot pri orodju EMMA
- končno poročilo naj bo izdelano v podobnem formatu in s podobnimi metrikami kot EMMA
- algoritem naj bo čim bolj optimiziran, glede na to, da je ta način analize pokritosti zelo računsko zahteven

V naslednjem poglavju tega diplomskega dela si bomo pogledali koncept mutacijskega testiranja. Tretje poglavje opisuje glavne tehnologije in orodja, ki smo jih uporabili pri razvoju

aplikacije. Četrto poglavje podrobno opisuje algoritem delovanja, arhitekturo aplikacije in način izvedbe. V petem poglavju so podana navodila za uporabo aplikacije ter primerjava natančnosti delovanja. Analizo pokritosti smo izvedli na štirih projektih in rezultate primerjali z rezultati, dobljenimi z orodjem EMMA. Na koncu še šesto poglavje, ki vsebuje zaključek in sklepne ugotovitve.

Poglavje 2 Mutacijsko testiranje

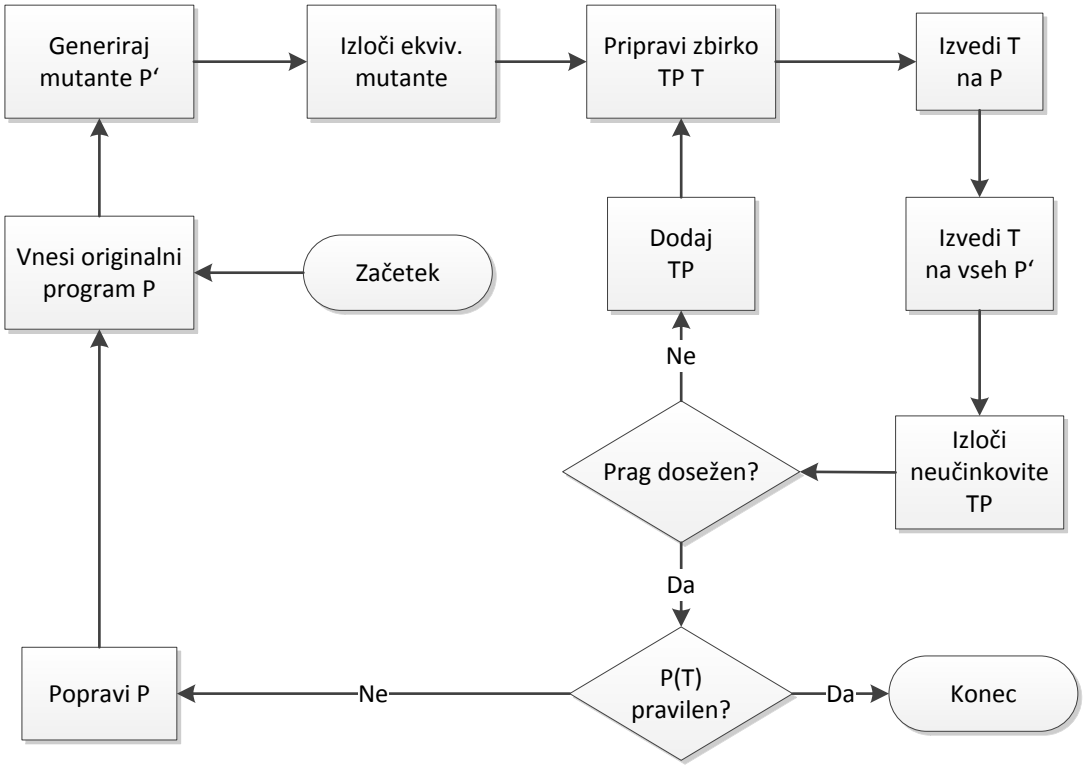
Mutacijsko testiranje ali tudi mutacijska analiza je metoda testiranja programske kode, ki se uporablja za ocenjevanje kvalitete obstoječih testov ter njihovo izboljšavo. Temelji na t.i. *mutacijah*, majhnih modifikacijah izvorne kode, ki jih namerno povzročamo v testirani programski kodi. Te modifikacije posnemajo najbolj pogoste napake, ki jih delajo programerji. Tako spremenjeno verzijo programske kode imenujemo *mutant*. Dobro zasnovane *testne zbirke* (angl. *test suite*) odkrijejo mutacijo v programski kodi in mutanta zavrnejo, čemur pravimo tudi, da mutanta ubijejo. Pri slabo zasnovanih ali pomanjkljivih testnih zbirkah pa lahko mutacije ostanejo neopažene. Kvaliteto testnih zbirk tako lahko ocenjujemo po deležu mutacij, ki so bile odkrite. Čim večji je ta delež, tem natančneje testna zbirka preveri testirano programsko kodo [3], [4].

Ideja metode mutacijskega testiranja je bila prvič predstavljena v sedemdesetih letih prejšnjega stoletja, od takrat je bilo vanjo vloženega veliko raziskovalnega dela z željo razviti jo v praktično uporabno metodo za testiranje programske opreme. Ker je metoda računsko zelo zahtevna, se zaradi visokih stroškov njene uporabe na začetku ni mogla uveljaviti v praksi, šele razvoj na področju strojne opreme in ogromna ponudba računske moči v zadnjem času sta omogočila njen razcvet [2].

Cilj testiranja je odkriti napake v programski kodi. Število potencialnih napak v programski kodi, ki jo testiramo, je ogromno in nemogoče je generirati mutante za vsako od njih. Zaradi tega se ideja mutacijskega testiranja omeji le na podmnožico vseh potencialnih napak in sicer tistih, ki so blizu pravilni verziji programske kode, z upanjem, da to zadošča za simulacijo vseh možnih napak. Ta pristop temelji na dveh hipotezah [2]:

- *hipoteza kompetentnega programerja* (angl. *Competent Programmer Hypothesis*), ki predpostavlja, da so programerji kompetentni, dobro usposobljeni za svoje delo in razvijajo programsko kodo, ki je blizu pravilni verziji. Čeprav so napake možne, so to zgolj enostavne napake, ki jih lahko popravimo z majhnimi sintaktičnimi korekcijami. Zaradi tega za mutacije izbiramo samo majhne sintaktične spremembe, kar predstavlja običajne napake, ki jih dela kompetenten programer.

- *hipoteza učinka združevanja* (angl. *Mutation Coupling Effect Hypothesis*), ki predpostavlja, da so kompleksni mutanti (to so mutanti, ki jih dobimo z več kot eno modifikacijo programske kode) sestavljeni iz enostavnih mutantov. Zaradi tega bo testna zbirka, ki odkrije vse enostavne mutante, odkrila tudi velik delež kompleksnih mutantov. Torej zadošča, da se pri mutacijskem testiranju omejimo na enostavne mutante.



Slika 1: Postopek mutacijskega testiranja

Slika 1 prikazuje postopek testiranja s pomočjo mutacijske analize. Iz originalnega programa *P* generiramo s pomočjo enostavnih sintaktičnih modifikacij množico defektnih programov *P′*, ki jih imenujemo *mutanti*. Naslednji primer prikazuje mutanta, ki ga dobimo tako, da v originalnem programu zamenjamo logični operator *and* (&&) z operatorjem *or* (||):

```
…
if (a && b) {
    c = 1;
} else {
    c = 0;
}
```

```
…
if (a || b) {
    c = 1;
} else {
    c = 0;
}
```

Transformacijsko pravilo, ki iz originalnega programa generira mutanta, imenujemo *mutacijski operator*. Tabela 1 prikazuje nekaj primerov mutacijskih operatorjev.

Mutacijski operator	Pomen	Razlaga
AOR	Arithmetic Operator Replacement	zamenjava aritmetičnega operatorja
LOR	Logical Operator Replacement	zamenjava logičnega operatorja
SDL	Statement DeLetion	brisanje stavka
VID	Variable Initialization Deletion	brisanje inicializacije spremenljivke
SVR	Scalar Variable Replacement	zamenjava skalarne spremenljivke
BSR	Bomb Statement Replacement	zamenjava stavka z bombo

Tabela 1: Primeri mutacijskih operatorjev

Množico mutantov lahko analiziramo s pomočjo hevrističnega algoritma, ki odkrije in poskuša izločiti čim več ekvivalentnih mutantov in na ta način zmanjša množico mutantov. V naslednji fazi izvajamo testne primere iz zbirke T, najprej na originalnem programu P in nato še na vsakem od mutantov P'. Če se rezultati izvajanja mutanta P' razlikujejo od rezultatov originalnega programa P za katerikoli testni primer iz zbirke T, takega mutanta označimo za mrtvega. Pravimo, da je testni primer mutanta ubit, v nasprotnem primeru pa, da je mutant preživel. Mrtve mutante zavržemo in jih ne uporabljamo v nadaljnjih izvajanjih. Testne primere, ki ne ubijejo vsaj enega mutanta, štejemo za neučinkovite in jih izločimo.

Ko izvedemo vse testne primere na mutantih, nam lahko ostane določeno število preživelih mutantov, ki jih noben test ni odkril. Poskusimo lahko izboljšati zbirko testov, dodati nove testne primere, vendar pa lahko naletimo na mutante, ki jih je nemogoče ubiti, ker vedno proizvedejo enake rezultate kot originalni program. Take mutante imenujemo *ekvivalentni mutanti* (angl. *equivalent mutants*). Sintaktično so drugačni kot izvorni program, funkcionalno pa so enakovredni. Avtomatska detekcija ekvivalentnih mutantov je nemogoča, ker gre za *neodločljiv* problem (angl. *undecidable*). Problem ekvivalentnih mutantov je ena od ovir za večjo razširjenost mutacijskega testiranja [1].

S pomočjo števila ubitih mutantov ter števila vseh ne-ekvivalentnih mutantov, lahko izračunamo *mutacijsko oceno* (angl. *mutation score*) po naslednji enačbi:

$$mutacijska\ ocena = \frac{število\ ubitih\ mutantov}{število\ vseh\ neekvivalentnih\ mutantov}$$

Mutacijska ocena pove, v kolikšni meri je testna zbirka odkrila mutacije v programski kodi in je torej mera za kvaliteto testne zbirke. Vrednost 1 pove, da so bili odkriti vsi mutanti oziroma vse napake v izvorni kodi. V praksi je tako vrednost zelo težko doseči, zato si izberemo prag,

to je minimalno sprejemljivo mutacijsko oceno. Če testna zbirka ne doseže praga, moramo dodati nove testne primer, s katerimi zajamemo še preživele mutante. Proces se nadaljujejo iterativno, dokler ne dosežemo zahtevanega praga.

Poglavje 3 Uporabljene tehnologije

V nadaljevanju so opisane glavne tehnologije, ki smo jih uporabili pri razvoju aplikacije.

3.1 EMMA

EMMA [6] je orodje za izvajanje analize pokritost javanske kode s testi, dostopno pod odprtokodno licenco CPL (Common Public License). EMMA deluje s pomočjo *instrumentacije* (angl. *instrumentation*) javanske vmesne kode (angl. byte code). Izraz instrumentacija pomeni, da v analizirano programsko kodo dodamo posebno kodo oziroma stavke, ki omogočajo spremljanje, diagnosticiranje in merjenje različnih parametrov med izvajanjem aplikacije. EMMA deluje na nivoju vmesne kode, tako da v vsak atomičen blok ukazov vstavi posebno zastavico, ki se postavi, ko oziroma če se pripadajoči blok izvede. Instrumentacijo lahko izvede v naprej, pred zagonom aplikacije, ali sproti, med nalaganjem javanskih razredov s pomočjo posebnega *nalagalnika razredov* (angl. *classloader*).

EMMA podpira naslednje metrike pokritosti kode: pokritost razredov, pokritost metod, pokritost osnovnih blokov in pokritost vrstic izvorne kode. Osnovni blok je zaporedje ukazov vmesne kode brez kakršnihkoli skokov ali ciljev skokov, ki se izvede atomarno, kar pomeni, da se izvede bodisi v celoti, bodisi sploh ne. Osnovni bloki so temeljna enota pri merjenju pokritosti kode. Izvajanje se beleži na nivoju osnovnega bloka, z drugimi besedami, za vsak osnovni blok se zabeleži, ali se je izvedel ali ne. Pokritost vrstice izvorne kode se oceni na osnovi pokritosti pripadajočih osnovnih blokov oziroma s projekcijo vrstice na pripadajoče osnovne bloke. Zaradi tega EMMA ne more vedno natančno ugotoviti pokritosti posamezne vrstice izvorne kode.

Poročilo o pokritosti kode, ki ga izdela EMMA, je lahko v treh formatih: HTML, XML ali tekstovno. Statistike pokritosti so podane na nivoju celotnega projekta ter za vsak javanski paket, razred in metodo posebej. Nastavimo lahko mejne vrednosti pokritosti in EMMA obarva z rdečo barvo tiste vrednosti, ki ne dosegajo postavljenega praga.

Pri diplomskem delu smo EMMO uporabili kot zgled delovanja za algoritem za analizo pokritosti kode, kot zgled pri izdelavi poročila ter na koncu za testiranje in primerjavo natančnosti delovanja.

3.2 Apache Maven

Apache Maven [7] je programsko orodje, namenjeno za upravljanje javanskih projektov in avtomatizacijo gradnje. Hkrati je tudi skupek standardov v procesu izdelave programske opreme. Maven temelji na konceptu *projektno objektnega modela* (angl. *Project Object Model*) oziroma s kratico POM. To je datoteka v formatu XML, ki vsebuje informacije o projektu ter konfiguracijske podatke, ki jih Maven potrebuje pri gradnji projekt. Model POM je hierarhičen. Obstaja t.i. super POM, to je korenski POM, v katerem so zapisane privzete vrednosti večine nastavitev, ki so na voljo. Od njega implicitno dedujejo vsi Maven projekti, direktno ali preko staršev, in tako podedujejo te nastavitve.

Njegove ključne značilnosti so:

- konvencija pred konfiguracijo: Maven se poskuša izogniti konfiguraciji, kolikor je le možno, z uporabo privzetih vrednosti, običajnih v realnem svetu ter v naprej pripravljenih predlog projektov s standardno strukturo.
- upravljanje odvisnosti: v datoteki POM definiramo odvisnosti projekta in pri gradnji jih Maven samodejno pridobi, skupaj s tranzitivnimi odvisnostmi.
- repozitorij ponovno uporabnih komponent: odvisnosti se lahko naložijo z lokalnega ali oddaljenega, privatnega ali javnega repozitorija. Obstaja tudi centralni repozitorij, kamor gre Maven privzeto iskat odvisnosti, ki jih ne najde lokalno.
- razširljivost preko vtičnikov: Maven je zasnovan modularno in je razširljiv preko vtičnikov.

Pri diplomskem delu smo Maven uporabili kot orodje za upravljanje projekta, poleg tega je aplikacija zasnovana kot vtičnik za Maven in je namenjena za analizo pokritosti Maven projektov.

3.3 Java Parser

Java Parser [8] je javanska knjižnica za razčlenjevanje javanske kode in generiranje *abstraktnega sintaksnega drevesa* AST (angl. *abstract syntax tree*). Dostopna je pod odprtokodno licenco LGPL. Omogoča kreiranje drevesa AST iz podane javanske kode, urejanje drevesa, urejanje obstoječih in dodajanje novih vozlišč, pretvorbo drevesa AST nazaj v

javansko izvorno kodo, sprehajanje po drevesu s pomočjo načrtovalskega vzorca »*Obiskovalec*« (angl. *visitor design pattern*). Je hitra in zmogljiva ter enostavna za uporabo.

Pri diplomskem delu smo knjižnico Java Parser uporabili za generiranje dreves AST za analizirane javanske datoteke, vstavljanje bomb v drevesa ter pretvorbo dreves nazaj v javansko (mutirano) izvorno kodo.

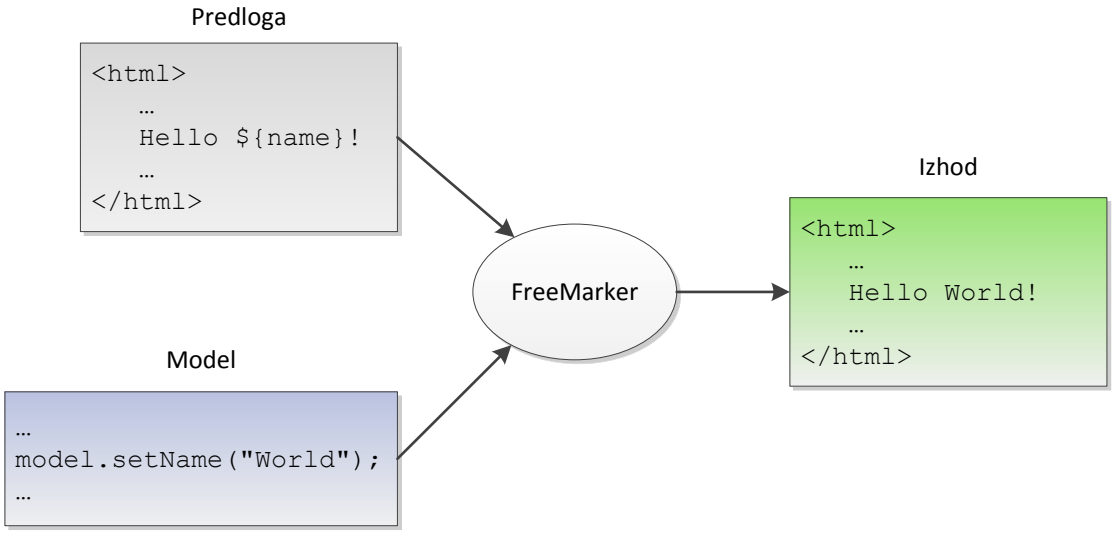
3.4 Hotswap Agent

Orodje Hotswap Agent [9] omogoča zamenjavo javanskih razredov v pomnilniku med delovanjem aplikacije (angl. hot-swap), brez ponovnega zagona aplikacije. Glavni cilj projekta je olajšati in pohitriti razvoj aplikacij tako, da se izognemo klasičnemu ciklu pri razvoju *spremeni → prevedi → ponovno zaženi in počakaj → preveri* ter ga nadomestimo z enostavnejšim in hitrejšim *spremeni → prevedi → preveri*. Hotswap Agent opazuje mapo, kjer se nahajajo prevedeni javanski razredi in ko zazna spremembo, spremenjene razrede ponovno naloži. Hotswap Agent ponuja tudi vtičnike za vsa glavna javanska ogrodja, ki zahtevajo posebne mehanizme za rekonfiguracijo aplikacije po zamenjavi razredov, npr. ogrodja Spring, Hibernate, Logback.

Pri diplomskem delu smo orodje Hotswap Agent uporabili pri izvajanju testov, da bi pohitrili delovanje aplikacije. Pri vsaki mutaciji smo tako lahko s pomočjo tega orodja zamenjali prevedeni javanski razred v pomnilniku in pognali teste v istem procesu.

3.5 Apache FreeMarker

Apache FreeMarker [10] je javanska knjižnica za delo s predlogami, ki omogoča, da na osnovi v naprej pripravljene predloge in poslovnih podatkov dinamično generiramo tekstovno vsebino, npr. HTML kodo, e-poštna sporočila, konfiguracijske datoteke. Osnovni princip delovanja je prikazan na sliki 2.



Slika 2: Apache FreeMarker

Za pisanje predlog uporabljamo poseben jezik, imenovan FreeMarker Template Language (FTL). Poslovne podatke, ki jih želimo vstaviti v predlogo, pripravimo v Javi ter jih podamo kot parameter pri procesiranju predloge. Ta pristop je poznan kot načrtovalski vzorec *model-pogled-krmilnik* (angl. *Model-View-Controller*) oziroma s kratico MVC, ki se je še posebej uveljavil pri načrtovanju spletnih aplikacij. Knjižnica Apache FreeMarker ni vezana na spletne aplikacije in strežniške tehnologije, ampak jo lahko uporabljamo tudi v samostojnih javanskih aplikacijah.

Pri diplomskem delu smo knjižnico Apache FreeMarker uporabili pri generiranju končnih poročil na osnovi v naprej pripravljenih predlog.

Poglavje 4 Arhitektura in implementacija

Aplikacijo za merjenje pokritosti kode s testi, ki smo jo razvili, smo poimenovali BBJCC, kar je angleška kratica za Bomb Based Java Code Coverage, kar bi lahko prevedli kot orodje za merjenje pokritosti javanske kode s testi na osnovi bomb. V tem poglavju bomo podrobno predstavili arhitekturo in zasnovo aplikacije, algoritem delovanja ter tehnično izvedbo.

4.1 Opis arhitekture

Aplikacija BBJCC je zasnovana kot vtičnik za orodje Maven in omogoča analizo Javanskih projektov strukturiranih kot Maven projekt, kar pomeni, da so pripravljeni za gradnjo (angl. *building*) z orodjem Maven in imajo strukturo map, skladno z Maven konvencijo. S pomočjo Mavena aplikacija pridobi vse potrebne poti (do mape z izvorno kodo in testnih razredov), seznam vseh knjižnic, potrebnih za prevajanje ter za poganjanje *testnih enot* (angl. *unit test*).

Aplikacija BBJCC je načrtovana kot Maven projekt z modularno zasnovo in je sestavljena iz naslednjih modulov:

- *bbjcc-maven-plugin*
- *bbjcc-utils*
- *bbjcc-test-samples*

Modul *bbjcc-maven-plugin* je vtičnik za Maven in predstavlja jedro aplikacije. Modul *bbjcc-utils* vsebuje pomožni Java razred, ki se uporablja pri mutacijski analizi izvorne kode. V svojem modulu se nahaja zato, ker se ta modul zapakira v *javanski arhiv* (datoteko JAR), ki se doda kot knjižnica na pot do razredov pri prevajanju mutirane izvorne kode in se kliče pri izvajanju testnih enot. Modul *bbjcc-test-samples* vsebuje zbirko javanskih razredov z vzorčno kodo za posamezne Javanske konstrukte in testnimi enotami ter služi za testiranje aplikacije ter primerjavo in umerjanje z orodjem EMMA.

4.2 Algoritem delovanja

Algoritem delovanja je prikazan na sliki 3 z diagramom poteka, opis sledi:

1. Aplikacijo poženemo kot vtičnik za Maven iz ukazne vrstice v korenski mapi obravnavanega projekta z ukazom *mvn*.

2. S pomočjo Mavena aplikacija pridobi vse potrebne poti, seznam odvisnosti za prevajanje izvorne kode in poganjanje testnih enot.

3. Algoritem za analizo pokritosti se sprehaja čez vse datoteke z Javansko izvorno kodo, razčleni izvorno kodo v posamezni datoteki in zgradi abstraktno sintaksno drevo (AST).

4. Algoritem se rekurzivno sprehaja po drevesu AST in na primerna mesta vstavlja bombe. Bomba je Javanski stavek, ki pri izvajanju povzroči izjemo, seveda le v primeru, če Java interpreter naleti nanjo pri izvajanju ukazov.

5. Drevo AST z vstavljeno bombo pretvorimo nazaj v izvorno kodo in dobimo t.i. mutirano izvorno kodo.

6. Javansko datoteko z mutirano izvorno kodo s pomočjo Java prevajalnika prevedemo v izvršljivo kodo (eno ali več datotek s končnico *class*).

7. Če je pri prevajanju prišlo do napake, takega mutanta zavržemo (bombo smo postavili na nedovoljeno mesto).

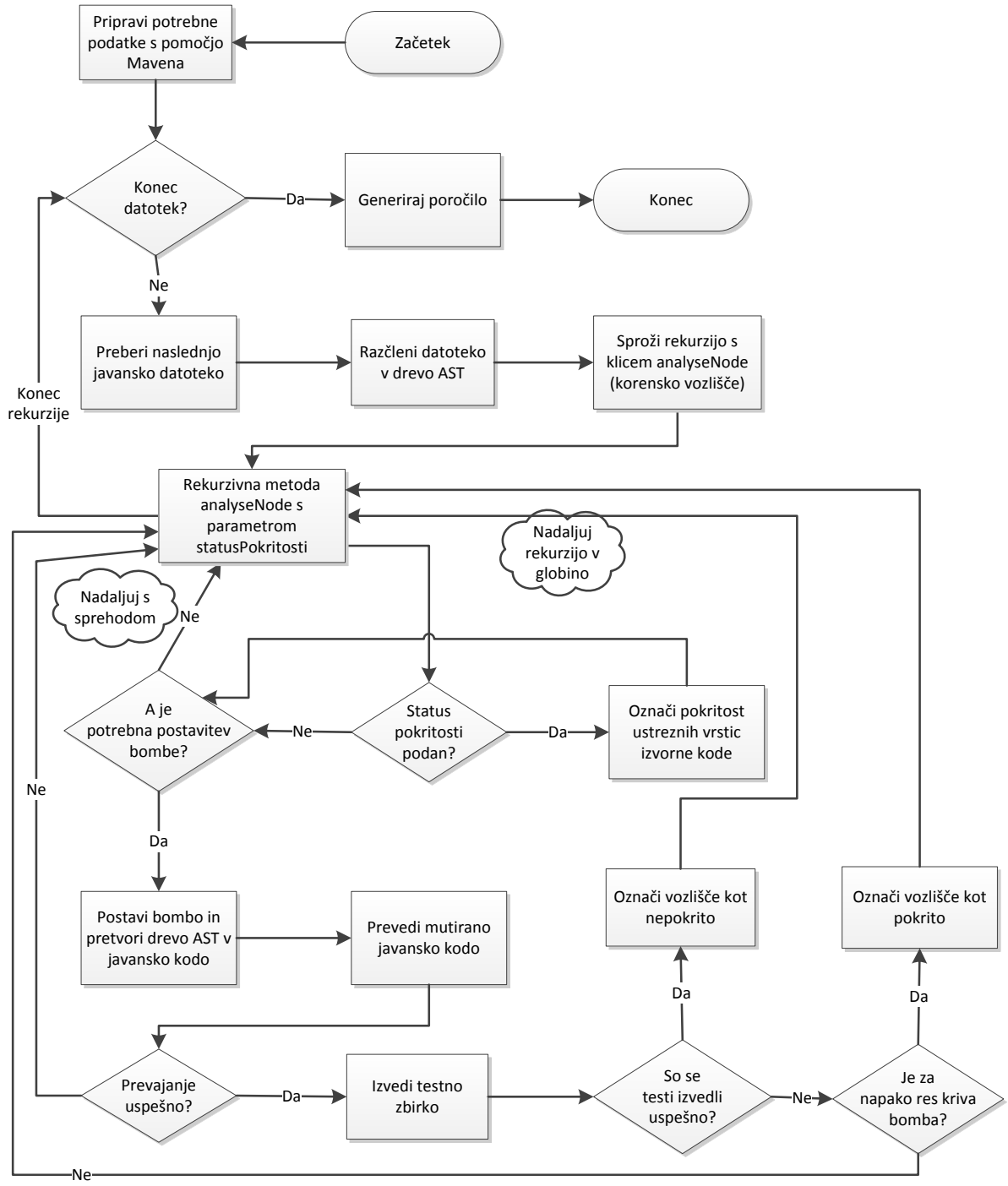
8. Izvedemo testne enote in preverimo, ali so se uspešno izvedle. Če je prišlo do napake, še preverimo, ali je napaka res posledica vstavljene bombe.

9. Če so se testi uspešno izvedli brez napake, lahko sklepamo, da Java interpreter ni naletel na vstavljeno bombo, stavek z bombo se ni izvedel. Torej lahko sklepamo, da se naslednji stavek v izvorni kodi tudi drugače ne bi izvedel, zato pripadajočo vrstico (eno ali več) označimo kot nepokrito.

10. Če je pri izvajanju testa prišlo do izjeme, ki je posledica vstavljene bombe, je Java interpreter naletel na bombo, torej lahko sklepamo, da bi se naslednji stavek za vstavljeno bombo izvedel, če v kodi ne bi bilo bombe. Če gre za enostaven stavek (npr. izraz), pripadajočo vrstico (eno ali več) označimo kot pokrito. Če pa gre za bolj kompleksen stavek (npr. kontrolni stavek), vemo samo, da bi se izvedel prvi korak (npr. preverjanje pogoja) in moramo rekurzivno nadaljevati s preiskovanjem. Kot pokritega lahko označimo samo ta prvi korak stavka.

11. Ko se sprehodimo čez celotno drevo AST, nadaljujemo z naslednjo datoteko z Javansko izvorno kodo.

12. Na koncu izdelamo poročilo z grafičnim prikazom pokritosti izvorne kode in povzetek po datotekah, paketih in za celoten projekt.



Slika 3: Diagram poteka algoritma za analizo pokritost kodi

4.3 Implementacija vtičnika za Maven

Vtičnik za Maven je ponovno uporabljiva Maven komponenta (angl. *artifact*), ki vsebuje deskriptor vtičnika in enega ali več MOJO razredov (angl. *Maven plain Old Java Object*). MOJO je Javanski razred, označen z označbo *@Mojo,* ki implementira vmesnik *Mojo* in predstavlja Maven cilj (angl. *goal*), to je akcijo, ki jo vtičnik izvede. Izbrano akcijo vtičnika poženemo iz ukazne vrstice z naslednjim ukazom:

```
mvn <ime vtičnika>:<cilj>
```

Vtičnik za Maven izdelamo kot običajni Maven projekt, ki ima v datoteki POM kot tip pakiranja navedeno vrednost *maven-plugin*:

```
<packaging>maven-plugin</packaging>
```

Po konvenciji vtičnik poimenujemo s pripono *-maven-plugin*, torej *bbjcc-maven-plugin*. Vtičnik *bbjcc-maven-plugin* vsebuje en Maven cilj - *analyse*, kateremu ustreza razred *BbjccMojo*:

```
@Mojo(name = "analyse")
public class BbjccMojo extends AbstractMojo
```

Cilj *analyse* vtičnika *bbjcc-maven-plugin* torej sprožimo iz ukazne vrstice z ukazom (*-maven-plugin* je standardna pripona in jo lahko izpustimo):

```
mvn bbjcc:analyse
```

MOJO mora implementirati metodo *execute*, definirano v vmesniku *Mojo*, ki se pokliče, ko poženemo ukaz v ukazni vrstici. Poleg tega je *Mojo* instanci na voljo metoda *getLog*, podedovana iz abstraktnega razreda *AbstractMojo*, ki vrne objekt *Log*, preko katerega vtičnik izpisuje sporočila o svojem delovanju:

```
void execute() throws MojoExecutionException, MojoFailureException;
Log getLog();
```

S pomočjo označbe *@Parameter* lahko zahtevamo, da Maven injicira zahtevano vrednost v podano spremenljivko, npr:

```
@Parameter(defaultValue = "${project.build.sourceDirectory}", readonly = true)
private File sourceDirectory;
```

Vtičnik *bbjcc-maven-plugin* za svoje delovanju potrebuje naslednje podatke, ki jih pridobi na ta način:

- *project.build.sourceDirectory*: mapa z izvorno kodo projekta, ki ga analiziramo (*src/main/java* v standardni Maven strukturi map)

- *project.build.directory*: izhodna mapa gradnje (*target* v standardni Maven strukturi map)

- *project.build.outputDirectory*: izhodna mapa, kamor prevajalnik shranjuje prevedene Java razrede (*target/classes*)

- *project.compileClasspathElements*: seznam poti do vseh knjižnic, ki jih prevajalnik potrebuje za prevajanje projekta. Ta seznam Maven pridobi iz datoteke *pom.xml* in vsebuje vse *odvisnosti* (angl. *dependency*), katerih *domet* (angl. *scope*) je enak *compile*.

- *project.testClasspathElements*: seznam poti do vseh knjižnic, ki so potrebne za prevajanje in izvajanje testnih razredov. Ta seznam Maven prav tako pridobi iz datoteke *pom.xml* in vsebuje vse odvisnosti, katerih domet je *test*.

V mapi *sourceDirectory* poiščemo vse datoteke z Javansko izvorno kodo, tako da se rekurzivno sprehodimo po strukturi map in iščemo datoteke s končnico *java*. Algoritem za ugotavljanje pokritosti kode se bo sprehodil čez ta seznam datotek in jih analiziral.

V mapi s testnimi razredi poiščemo seznam vseh testnih enot, ki jih potrebujemo za odkrivanje mutacij. Če se omejimo na najbolj razširjeno Javansko orodje za testiranje enot JUnit, jih najdemo na dva načina:

- pri JUnit 4 so testne metode označene z označbo *@Test*. Testne razrede torej najdemo tako, da preiščemo mapo s testi (*target/test-classes* v standardni Maven strukturi map) in iščemo vse razrede, ki imajo vsaj eno tako metodo. Za to nalogo smo uporabili knjižnico Annotation Detector[1].

- pri JUnit 3 testni razredi razširjajo abstraktni razred *TestCase*. Sprehoditi se torej moramo po mapi s testi in poiskati vse take razrede.

Pripravimo še *pot do razredov* (angl. *classpath*), ki bo potrebna pri prevajanju mutiranih izvornih datotek. To pot dobimo tako, da združimo poti v seznamu *compileClasspathElements* z ustreznim ločilom (dvopičje ali podpičje, odvisno od operacijskega sistema) v niz ter dodamo še knjižnico *bbjcc-utils* (modul aplikacije BBJCC), ki vsebuje pomožne metode za mutacijsko

[1] https://github.com/rmuller/infomas-asl

testiranje. Ta knjižnica (datoteka JAR) se pri namestitvi aplikacije BBJCC namesti v lokalni Maven repozitorij, pot do nje pa dobimo s pomočjo instance *PluginDescriptor*.

V mapi *project.build.directory* pripravimo mapo *site/bbjcc*, kamor se bo shranilo končno poročilo analize pokritosti.

Ko je vse to pripravljeno, kreiramo instanco razreda *CoverageAnalyser* in poženemo analizo.

4.4 Postavljanje »bomb«

Za kreiranje bomb smo uporabili stavek *assert*, ki se v Javi uporablja za preverjanje domnev o delovanju programa. Stavek *assert* za bombo ima kot logični izraz uporabljeno logično vrednost *neresnično* (*false*):

```
assert false : "sporočilo o napaki";
```

Ta stavek pri izvajanju povzroči izjemo *AssertionError*, ki jo običajni *try-catch* stavki ne ujamejo in lahko predpostavljamo, da bo taka izjema res povzročila napako pri izvajanju testnih enot.

Predpogoj za uporabo stavka *assert* kot bombe pa je, da so *trditve* (angl. *assertions*) pri izvajanju aplikacije omogočene, privzeto so namreč onemogočene. Trditve omogočimo s stikalom *-enableassertions* ali *-ea* v ukazni vrstici programa *java*:

```
java –ea …
```

Za razliko od stavka *assert* pa stavek *throw* ne bi bil primerna izbira za postavljanje bomb. Če bi npr. analizirali metodo *method2* v naslednjem primeru in znotraj te metode postavili bombo:

```
void method1() {
    try {
        method2();
    } catch (Exception e) {
        logger.error("Prišlo je do napake.", e);
    }
}
void method2() {
    …
    throw new Exception("bomb");
    …
}
```

Tako postavljena bomba ne bi povzročila napake pri izvajanju testov, saj bi stavek *catch* ujel in obravnaval izjemo, sproženo z bombo.

Algoritem za analizo pokritosti uporablja bombe dveh vrst:

- kot stavek
- znotraj logičnega izraza

Bombo kot stavek naredimo na naslednji način:

```
mutationsCounter++;
Expression msgExpression = new StringLiteralExpr("mutation_" + mutationsCounter);
AssertStmt assertStmt = new AssertStmt(new BooleanLiteralExpr(false),
                                       msgExpression);
```

Bombo postavljamo vedno znotraj stavčnega bloka:

```
{
    …
    assert false : "mutation_i";
    …
}
```

Stavku *assert* dodamo sporočilo o napaki »*mutation_i*«, kjer je *i* zaporedna številka mutacije. Tako lahko pri spodletelih testnih enotah preverimo, če je vzrok za napako res podstavljena bomba in ne kakšna druga, nepredvidena napaka.

Postavljanje bombe v logični izraz je nekoliko bolj zapleteno. To funkcionalnost potrebujemo npr. pri analizi pokritosti stavka *if*. Stavka *assert* ne moremo postaviti direktno v logični izraz, zato smo si pomagali s pomožno metodo *assertFalse*, ki vsebuje bombo:

```
public class CoverageAnalyserUtils {
    public static boolean assertFalse(String message) {
        assert false : message;
        return true;
    }
}
```

Ta metoda vrača logično vrednost, zato lahko njen klic uporabimo kot logični izraz, npr:

```
if (bbjcc.utils.CoverageAnalyserUtils.assertFalse("mutation_i")) {
    …
```

Logični izraz s klicem metode naredimo na naslednji način:

```
mutationsCounter++;
NameExpr nameExpr = new NameExpr("bbjcc.utils.CoverageAnalyserUtils");
Expression msgExpression = new StringLiteralExpr("mutation_" + mutationsCounter);
Expression bombExpression = new MethodCallExpr(nameExpr, "assertFalse",
                            Arrays.asList(msgExpression));
```

Razred *CoverageAnalyserUtils* se mora nahajati na poti do razredov med prevajanjem mutirane kode in med izvajanjem testnih enot. Zato se ta razred nahaja v posebnem modulu aplikacije BBJCC z imenom *bbjcc-utils*, ki se zapakira v datoteko JAR. To datoteko potem dodamo kot knjižnico na pot do razredov.

Bombe postavljamo na pet mest v drevesu AST (pri čemer mislimo na nivo podatkovnih struktur znotraj drevesa AST):

- v stavčni blok
- v stavek *switch*
- v pogoj stavka *if*
- v pogoj stavka *for*
- v pogoj stavka *do-while*

Na nivoju javanske izvorne kode pa bombe postavljamo v vse tipe javanskih stavkov. Razlika je v tem, da ko npr. bombo postavimo v javanski stavek *if*, ga na nivoju drevesa AST dejansko postavimo v telo stavka, ki je tipa stavčni blok oziroma *BlockStmt*.

4.5 Preiskovanje drevesa AST

V zanki se sprehajamo čez vse datoteke z Javansko izvorno kodo (datoteke s končnico *java*), razčlenimo izvorno kodo v posamezni datoteki in zgradimo abstraktno sintaksno drevo (AST). Za to nalogo smo uporabili odprtokodno orodje JavaParser [8]. Metoda *parse* prebere javansko izvorno kodo iz vhodnega toka in vrne objekt *CompilationUnit*:

```
FileInputStream in = new FileInputStream(javaFile);
CompilationUnit cu;
try {
    cu = JavaParser.parse(in);
} finally {
    in.close();
}
```

Objekt *CompilationUnit* predstavlja korensko vozlišče drevesa AST, od tukaj začnemo rekurzivno sprehajanje po drevesu. Najprej preberemo ime paketa, v katerem se nahaja izbrana Java datoteka:

```
String packageName = cu.getPackage().getPackageName();
```

V podatkovni strukturi *CoverageData*, kamor shranjujemo surove podatke o pokritosti kode, pripravimo prostor za obravnavano Java datoteko:

```
coverageData.addJavaFileData(javaFile, new JavaFileData(javaFile, packageName,
numOfLines));
```

Na koncu, ko bo analiza pokritosti kode končana, bomo s pomočjo teh podatkov generirali poročilo. Preiskovanje nadaljujemo pri otrocih vozlišča *cu*:

```
for (Node node : cu.getChildrenNodes()) {
    …
```

Otroci korenskega vozlišča *cu* so lahko:

- razred (*class*)
- vmesnik (*interface*)
- naštevni podatkovni tip (*enum*)

Če gre za razred ali naštevni podatkovni tip, nadaljujemo s preiskovanjem, v primeru vmesnika pa zaključimo, saj vmesnik nima izvršljive kode.

Preiskovanje poteka rekurzivno, tako da kličemo rekurzivno metodo *analyseNode*:

```
void analyseNode(CompilationUnit cu, File javaFile, String classFQN, Node node,
CoverageStatus coverageStatus)
```

Metoda *analyseNode* sprejme naslednje parametre:

- *cu*: objekt *CompilationUnit* drevesa AST, ki ga trenutno preiskujemo
- *javaFile*: Java datoteka
- *classFQN*: polno kvalificirano ime javanskega razreda, kjer se trenutno nahajamo (ime paketa + ime razreda)
- *node*: vozlišče drevesa AST, ki ga analiziramo
- *coverageStatus*: pokritost kode za obravnavano vozlišče

CoverageStatus je naštevni podatkovni tip in lahko zavzame naslednje vrednosti:

- *COVERED*

- NOT_*COVERED*
- *UNDEFINED*

Vrednost parametra *coverageStatus* predstavlja rezultat analize predhodnega vozlišča. Npr. ko analiziramo blok stavkov, vstavljamo bombe na mesta med posameznimi stavki. Glede na to, ali se testi enot izvedejo uspešno, sklepamo na pokritost naslednjega stavka (vozlišča) v bloku. Ko pokličemo metodo *analyseNode* za naslednje vozlišče, to vrednost podamo kot parameter *coverageStatus*. Delovanje metode *analyseNode* je odvisno od vrednosti tega parametra:

- Če je vrednost enaka *COVERED*, vemo, da se stavek (pri enostavnih vozliščih) oziroma vsaj prvi korak stavka (pri kompleksnih vozliščih) izvede in ga lahko označimo kot pokritega. Pri kompleksnih vozliščih rekurzivno nadaljujemo pri otrocih vozlišča, vrednosti *coverageStatus* pri teh nadaljnjih klicih ne poznamo.
- Če je vrednost enaka NOT_*COVERED*, vemo, da se stavek ne izvede in ga lahko označimo kot nepokritega. Pri kompleksnih vozliščih rekurzivno nadaljujemo pri otrocih vozlišča, vrednost *coverageStatus* je pri teh nadaljnjih klicih prav tako NOT_*COVERED*.

Kot enostavno vozlišče mislimo vozlišče, katerega otroci niso samostojni stavki, ampak izrazi, npr. klic metode, prireditveni stavek, *return* stavek. Kot kompleksno vozlišče pa mislimo stavek, katerega otroci so samostojni stavki, npr. vejitveni stavek, zanka, definicija metode.

Metoda *analyseNode* ni generična, ampak je za vsako vrsto Javanskih stavkov pripravljena posebna (preobložena) metoda. V Javi se izbira, katera od preobloženih metod se bo poklicala, izvrši že v času prevajanja. Če torej pokličemo metodo:

```
analyseNode(cu, javaFile, classFQN, (Node)node, coverageStatus)
```

se bo izvedla tista od preobloženih metod, ki ima v deklaraciji kot tip parametra *node* naveden razred *Node*. Zaradi tega potrebujemo generično metodo *analyseNode* s parametrom *node* tipa *Node*, ki v *switch* stavku ugotovi dejanski tip spremenljivke s pomočjo operatorja *instanceof* ter pokliče ustrezno preobloženo metodo. V nadaljevanju za vsak tip vozlišča navajamo logiko za analizo takega tipa vozlišč.

4.5.1 Razred

Tip vozlišča v drevesu AST: *ClassOrInterfaceDeclaration*

Tip *ClassOrInterfaceDeclaration* vključuje razrede in vmesnike. Če gre za vmesnik, zaključimo, saj vmesnik nima izvršljive kode in ne moremo vstavljati bomb.

Pri začetnem rekurzivnem klicu določimo polno kvalificirano ime razreda in ga podamo kot parameter *classFQN*. Pri nadaljnjem preiskovanju drevesa AST se lahko zgodi, da pridemo v gnezdeni razred. V tem primeru moramo določiti novo polno kvalificirano ime razreda, kar

naredimo tako, da imenu starševskega razreda dodamo znak '$' in ime notranjega razreda. To ime se potem uporablja v nadaljnjih rekurzivnih klicih.

Razred dodamo v podatkovno strukturo *coverageData*, ki zbira podatke za končno izdelavo poročila. Nato se v zanki sprehodimo po vseh otrocih vozlišča (razreda) in rekurzivno kličemo metodo *analyseNode* za vsakega.

Na koncu moramo razred ponovno prevesti, tako da imamo na disku originalno verzijo prevedene *class* datoteke, ko nadaljujemo z analizo drugih razredov. Po vsakem ciklu mutacija/prevajanje/testi na disku namreč ostane mutirana verzija datoteke. Iz drevesa AST mutacijo (bombo) vedno odstranimo, vendar ga pred začetkom nove mutacije ne prevajamo, ker je to nepotrebno in časovno potratno.

4.5.2 Naštevni podatkovni tip

Tip vozlišča v drevesu AST: *EnumDeclaration*

Logika je podobna kot pri razredu.

4.5.3 Metoda in konstruktor

Tip vozlišča v drevesu AST: *MethodDeclaration, ConstructorDeclaration*

Za telo metode, ki je tipa *BlockStmt*, rekurzivno pokličemo metodo *analyseNode*. Metoda vrne stopnjo pokritosti stavčnega bloka (*enum* vrednost *BlockCoverage*). Na osnovi stopnje pokritosti označimo pokritost zaključnega zavitega oklepaja, skladno z orodjem EMMA. Če je stopnja pokritosti polna (to pomeni, da se izvede tudi bomba, postavljena na konec bloka), označimo zaključni oklepaj kot pokrit. V nasprotnem primeru ga označimo kot nepokritega, razen če je zadnji stavek v bloku *return* ali *throw*, v tem primeru ga pustimo neoznačenega.

Na koncu še povečamo števec metod obravnavanega razredu v podatkih za izdelavo poročila in zraven zabeležimo, ali se je metoda izvedla (oziroma bila poklicana). Če je stopnja pokritosti bloka polna ali delna, lahko sklepamo, da se je metoda res izvedla.

4.5.4 Stavčni blok

Tip vozlišča v drevesu AST: *BlockStmt*

Sintaksa stavka:

```
{
    stavek1;
    stavek2;
    …
    stavekn;
}
```

Stavčni blok je najpomembnejše mesto za postavljanje bomb. Nastopa lahko kot samostojen stavek ali kot del (telo) stavkov *if*, *switch*, *while*, *do-while*, *for*, *for-each*, *try-catch*, *synchronized* ter kot telo metod in konstruktorjev.

Vrednost parametra *coverageStatus* nam pove, ali je začetek stavčnega bloka pokrit (torej ali se sploh kdaj začne izvajati). Če ni (vrednost parametra je NOT_*COVERED*), potem nima smisla postavljati bomb, ampak se samo sprehodimo po vseh stavkih stavčnega bloka, rekurzivno kličemo metodo *analyseNode* za vsak stavek z vrednostjo parametra *coverageStatus* prav tako enako NOT_*COVERED* in označimo vse stavke kot nepokrite.

V nasprotnem primeru, če je vrednost parametra enaka *COVERED* ali če ni znana, moramo podrobno raziskati pokritost stavčnega bloka s postavljanjem bomb med stavki. Na osnovi dejstva, da javanski interpreter vstopi v stavčni blok, lahko sklepamo samo to, da je pokrit prvi stavek v bloku, za nadaljnje stavke pa ne vemo. Možno je recimo, da pri določenem stavku vedno pride do izjeme in se vsi nadaljnji stavki nikoli ne izvršijo.

V zanki se sprehodimo čez vse stavke stavčnega bloka in postavljamo bombe na vsa mesta med stavki, na začetek bloka ter na konec bloka (če je vrednost parametra *coverageStatus* enaka *COVERED*, nam na začetek bloka ni treba). Bombo, ki je tipa *Statement*, vrinemo na ustrezno mesto v drevo AST:

```
Statement bombStmt = createBombStatement();
blockStmt.getStmts().add(i, bombStmt);
```

Ko je bomba postavljena, izvedemo testne enote. Na osnovi rezultata testov sklepamo na pokritost naslednjega stavka v seznamu stavkov stavčnega bloka. Če je bomba odkrita, sklepamo, da bi se naslednji stavek izvedel, če le te ne bi bilo ter zanj rekurzivno pokličemo metodo *analyseNode*. S tem klicem označimo naslednji stavek kot pokrit oziroma ga nadalje analiziramo, če gre za kompleksen stavek. Npr. če gre za stavek *if*, lahko označimo kot pokrito

samo vrstico s pogojem, za ostale vrstice pa moramo stavek nadalje analizirati z dodatnimi bombami.

Če postavljena bomba v stavčnem bloku ni bila odkrita, sklepamo, da naslednji stavek kot tudi vsi nadaljnji stavki v istem bloku niso pokriti – se nikoli ne izvedejo, zato lahko prenehamo s postavljanjem bomb. Namesto tega se samo sprehodimo do konca bloka ter vse nadaljnje stavke označimo kot nepokrite s pomočjo rekurzivnega klica metode *analyseNode* s parametrom *coverageStatus* postavljenim na *NOT_COVERED*.

Metoda *analyseNode* za stavčni blok vrne pokritost bloka, to je vrednost tipa B*lockCoverage*, ki pove, v kolikšni meri se podani blok izvrši. Možne vrednosti so:

- polna pokritost (*FULL*) blok se izvrši do konca, odkrita je tudi bomba, postavljena za zadnjim stavkom bloka. Blok, ki se zaključi s stavkom *return* ali *throw*, ne more doseči polne pokritosti.
- delna pokritost (*PARTIAL*): izvrši se vsaj prvi stavek v bloku
- nična pokritost (*NONE*): blok se sploh ne začne izvajati

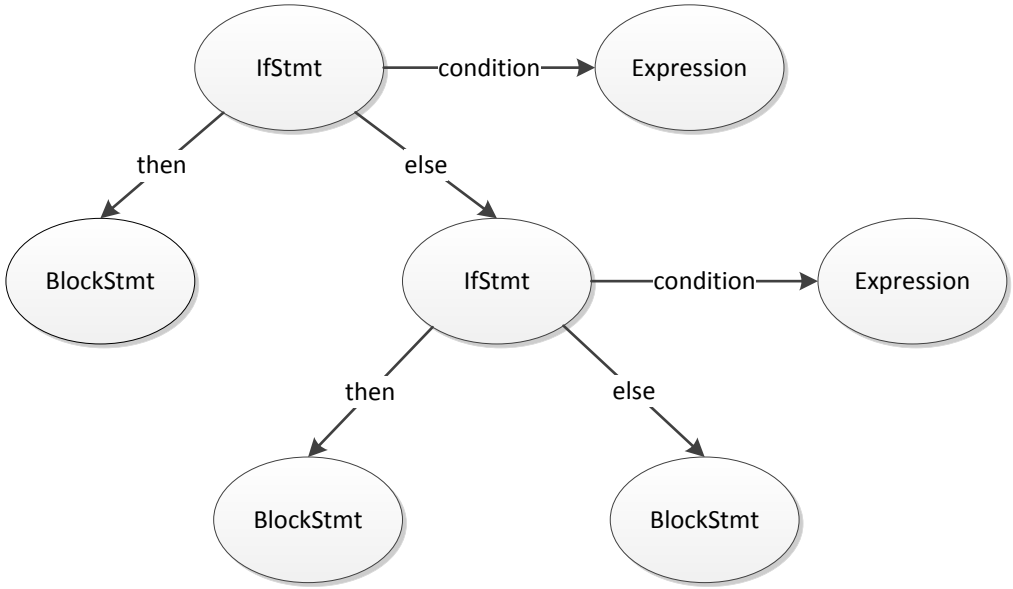
Pokritost bloka potrebujemo za označitev zaključnega zavitega oklepaja bloka, v skladu z orodjem EMMA.

4.5.5 Stavek If

Tip vozlišča v drevesu AST: *IfStmt*

Primer stavka *if* ter pripadajoče drevo AST, ki je prikazano na sliki 4:

```
if (pogoj1) {
    blok1;
} else if (pogoj2) {
    blok2;
} else {
    blok3;
}
```



Slika 4: Drevo AST za stavek *if – else if - else*

Pri analizi stavka *If* si pomagamo z bombo, ki jo vstavimo v pogojni izraz (pogoj *if* ter pogoj *else if*). Na ta način ugotovimo, ali se je obravnavani pogoj dejansko izvedel in ga lahko označimo kot pokritega / nepokritega. Možno namreč je, da se pogoj izvede, vendar nima nikoli logične vrednosti *true*, zato se pripadajoči stavek nikoli ne izvede. Samo z vstavljanjem bombe v stavčni blok tega ne bi mogli ugotoviti. Prejšnji primer stavka *If* z bombo v pogojnem izrazu:

```
if (pogoj1) {
    blok1;
} else if (bbjcc.utils.CoverageAnalyserUtils.assertFalse("mutation_6")){
    blok2;
} else {
…
```

Nadaljujemo z analizo veje *then*. Če je stavek *then* zapisan z zavitimi oklepaji, je pripadajoče vozlišče tipa *BlockStmt* in ga lahko direktno uporabimo v rekurzivnem klicu *analyseNode* (bombe lahko postavljamo samo v stavčne bloke). Sintaktično pa je pravilno tudi brez zavitih oklepajev:

```
if (pogoj)
    stavek1;
```

V tem primeru stavek *then* ni tipa *BlockStmt* in ga moramo najprej pretvoriti v *BlockStmt*, preden lahko nadaljujemo z rekurzijo. To naredimo na naslednji način:

```
Statement thenStmt = ifStmt.getThenStmt();
BlockStmt blockStmt = new BlockStmt();
blockStmt.getStmts().add(thenStmt);
```

```
ifStmt.setThenStmt(blockStmt);
```

Nato pokličemo metodo *analyseNode* ter na koncu povrnemo drevo AST v originalno stanje:

```
analyseNode(cu, javaFile, classFQN, (BlockStmt) ifStmt.getThenStmt());
ifStmt.setThenStmt(thenStmt);
```

Nadaljujemo z vejo *else*. Če gre za stavek *else if*, je vozlišče tipa *IfStmt*, v tem primeru nadaljujemo z rekurzivnim klicem metode *analyseNode*. Če je vozlišče tipa *BlockStmt*, prav tako nadaljujemo s klicem *analyseNode*. V nasprotnem primeru pa moramo stavek *else* podobno kot pri veji *then* najprej pretvoriti v *BlockStmt*.

4.5.6 Stavek Switch

Tip vozlišča v drevesu AST: *SwitchStmt*

Sintaksa stavka:

```
switch (izraz) {
    case vrednost1:
        stavki
        break; (opcijsko)
    case vrednost2:
        stavki
        break; (opcijsko)
    default: (opcijsko)
        stavki
}
```

Najprej označimo selektor stavka *switch* glede na vrednost parametra c*overageStatus*. Nato se v zanki sprehodimo po vseh vejah, ki so tipa *SwitchEntryStmt*. Za vsako vejo dobimo seznam stavkov:

```
List<Statement> stmts = switchEntryStmt.getStmts();
```

V zanki se sprehodimo čez vse stavke. Če stavek *switch* ni pokrit (vrednost parametra *coverageStatus* je NOT_*COVERED*), samo označimo vsak stavek kot nepokrit z rekurzivnim klicem metode *analyseNode* s parametrom *coverageStatus* enakim NOT_*COVERED*. V nasprotnem primeru pa podobno kot pri analizi stavčnega bloka postavljamo bombe na vsa mesta v seznamu *stmts* (razen na konec seznama, kar ni smiselno), izvajamo testne enote in na osnovi rezultata testov sklepamo na pokritost naslednjega stavka v seznamu ter rekurzivno kličemo metodo *analyseNode* za vsak ta naslednji stavek. Če ugotovimo, da nek stavek ni pokrit, sklepamo, da tudi vsi naslednji v istem bloku niso, zato za vse naslednje samo pokličemo metodo *analyseNode* s parametrom *coverageStatus* enakim NOT_*COVERED*.

4.5.7 Stavek While

Tip vozlišča v drevesu AST: *WhileStmt*

Sintaksa stavka:

```
while (pogoj)
    telo
```

Najprej označimo vrstico s pogojem stavka *while* glede na vrednost parametra c*overageStatus* kot pokritega / nepokritega.

Telo stavka *while* je lahko stavčni blok (tip vozlišča *BlockStmt*) ali stavek. V prvem primeru enostavno rekurzivno pokličemo metodo *analyseNode*, v drugem primeru pa moramo telo najprej pretvoriti v stavčni blok ter po klicu *analyseNode* pretvoriti nazaj v originalno stanje.

4.5.8 Stavek Do-While

Tip vozlišča v drevesu AST: *DoStmt*

Sintaksa stavka:

```
do
    telo
while (pogoj);
```

Pri stavku *do-while* je za razliko od stavka *do* pogoj na koncu in v naprej ne vemo, ali je pokrit ali ne. Telo stavka *do-while* je lahko stavčni blok (tip vozlišča *BlockStmt*) ali stavek. V prvem primeru enostavno rekurzivno pokličemo metodo *analyseNode*, v drugem primeru pa moramo telo najprej pretvoriti v stavčni blok ter po klicu *analyseNode* pretvoriti nazaj v originalno stanje.

Pokritost pogoja ugotovimo tako, da postavimo bombo v obliki logičnega izraza v sam pogoj:

```
do {
    stavčni blok
} while (bbjcc.utils.CoverageAnalyserUtils.assertFalse("mutation_i"));
```

4.5.9 Stavek For

Tip vozlišča v drevesu AST: *ForStmt*

Sintaksa stavka:

```
for (inicializacija; pogoj; inkrement)
    telo
```

Elementi inicializacija, pogoj in inkrement niso obvezni, zato moramo preveriti, ali so definirani.

Na osnovi vrednosti parametra c*overageStatus* sklepamo na pokritost inicializacijskega izraza in pogoja ter ju ustrezno označimo kot pokrita / nepokrita (v stavku *for* se najprej izvede inicializacija, takoj nato se preveri pogoj).

Pokritost inkrementalnega izraza ugotovimo s pomočjo bombe, ki jo dodamo v sam izraz s pomočjo operatorja vejica, npr:

```
for (int i = 0; i < 10; i++, bbjcc.utils.CoverageAnalyserUtils
                            .assertFalse ("mutation_i")) {
```

Bombo dodamo v inkrementalni izraz na naslednji način:

```
Expression bombExpression = createBombExpression();
forStmt.getUpdate().add(bombExpression);
```

Nadaljujemo z ugotavljanjem pokritosti telesa stavka *for*. Telo je lahko stavčni blok (tip vozlišča *BlockStmt*) ali stavek. V prvem primeru enostavno rekurzivno pokličemo metodo *analyseNode*, v drugem primeru pa moramo telo najprej pretvoriti v stavčni blok ter po klicu *analyseNode* nazaj v originalno stanje.

4.5.10 Stavek For-Each

Tip vozlišča v drevesu AST: *ForeachStmt*

Sintaksa stavka:

```
for (element : zbirka)
    telo
```

Pokritost prve vrstice označimo na osnovi vrednosti parametra c*overageStatus*.

Nadaljujemo z ugotavljanjem pokritosti telesa stavka *for-each*. Podobno kot pri stavku *for* je telo lahko stavčni blok (tip vozlišča *BlockStmt*) ali stavek. V prvem primeru enostavno rekurzivno pokličemo metodo *analyseNode*, v drugem primeru pa moramo telo najprej pretvoriti v stavčni blok ter po klicu *analyseNode* nazaj v originalno stanje.

4.5.11 Stavek Try-Catch

Tip vozlišča v drevesu AST: *TryStmt*

Sintaksa stavka:

```
try {
    stavčni blok
} catch (…) {
    stavčni blok
} finally {
    stavčni blok
}
```

Najprej preverimo pokritost stavčnega bloka *try*. Vozlišče je tipa *BlockStmt*, zato enostavno rekurzivno pokličemo metodo *analyseNode.* Stavek *try-catch* ima lahko več blokov *catch*, zato se v zanki sprehodimo čez vse in pokličemo *analyseNode* za vsakega od njih. Glede na status pokritosti bloka *catch*, ki jo vrne metoda, označimo še pokritost pripadajoče vrstice *catch* z deklaracijo izjeme. Na koncu na enak način preverimo še pokritost bloka *finally*.

4.5.12 Stavek Label

Tip vozlišča v drevesu AST: *LabeledStmt*

Sintaksa stavka:

```
label:
stavek
```

Rekurzivno pokličemo metodo *analyseNode* na pripadajočem stavku.

4.5.13 Stavek synchronized

Tip vozlišča v drevesu AST: *SynchronizedStmt*

Sintaksa stavka:

```
synchronized (…) {
    stavčni blok
}
```

Najprej označimo pokritost prve vrstice na osnovi vrednosti parametra c*overageStatus.* Nadaljujemo z analizo stavčnega bloka, ki je tipa *BlockStmt*, tako da rekurzivno pokličemo metodo *analyseNode*.

4.5.14 Stavek return

Tip vozlišča v drevesu AST: *ReturnStmt*

Sintaksa stavka:

```
return izraz;
```

Stavek *return* lahko v izrazu vsebuje definicijo anonimnega razreda, zato se moramo rekurzivno sprehoditi po pripadajočem drevesu in stavek raziskati v globino. EMMA šteje anonimne razrede k skupnemu številu razredov ter njihove metode k skupnemu številu metod v nadrejenem razredu.

4.5.15 Enostavni tipi stavkov

Enostavni tipi stavkov so:

- izraz (*ExpressionStmt*)
- stavek *break* (*BreakStmt*)
- stavek *continue* (*ContinueStmt*)
- stavek *assert* (*AssertStmt*)
- stavek *throw* (*ThrowStmt*)

Vozlišča teh tipov nimajo nobenih podrejenih vozlišč tipa stavek (*Statement*), zato ni potrebno nadaljevati z rekurzijo, ampak samo označimo pokritost ustrezne vrstice / vrstic na osnovi vrednosti parametra c*overageStatus*.

4.6 Pokritost vejitev

Analizo pokritosti vejitev smo implementirali za stavka *if* in *switch*. S pomočjo postavljenih bomb lahko ugotovimo, koliko vejitev v teh dveh pogojnih stavkih se dejansko izvede in iz tega izračunamo pokritost vejitev za izbrani stavek ter skupno za celotni javanski razred.

Pri stavku *if* sta možni dve vejitvi:

- pogoj je izpolnjen (vrednost pogojnega izraza je *true*)
- pogoj ni izpolnjen (vrednost pogojnega izraza je *false*)

Pokritost posameznega stavka *if* je torej lahko:

- 0/2 (0 %): stavek *if* se sploh ni izvedel in s tem tudi ne nobena veja
- 1/2 (50 %): samo ena od vej se je izvedla

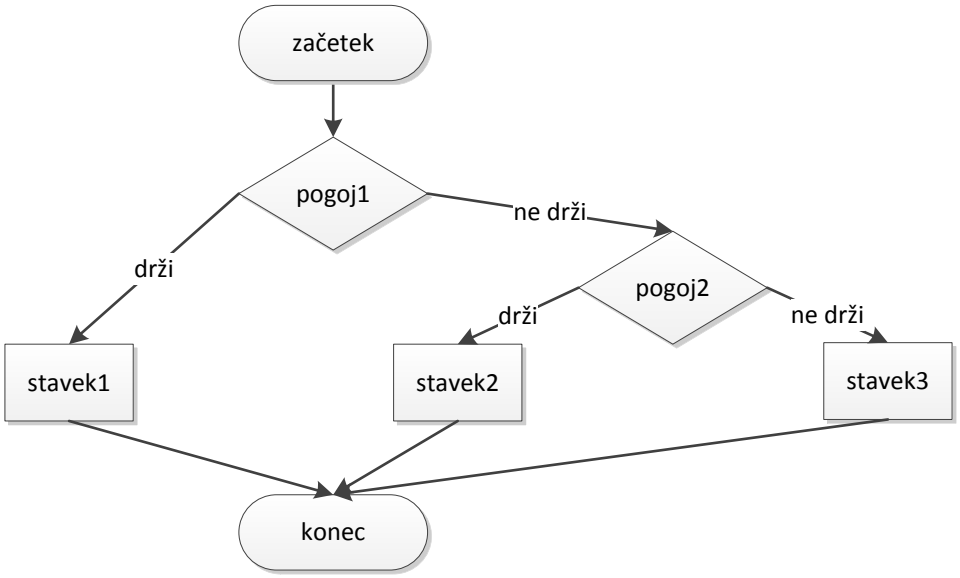
- 2/2 (100 %): obe veji sta se izvedli

Stavek *else-if* obravnavamo kot nov stavek *if*. Naslednja dva zapisa sta enakovredna:

```
if (pogoj1) {
    stavek1;
} else if (pogoj2) {
    stavek2;
} else {
    stavek3;
}

if (pogoj1) {
    stavek1;
} else {
    if (pogoj2) {
        stavek2;
    } else {
        stavek3;
    }
}
```

Prvi zapis je krajša in bolj pregledna oblika drugega zapisa ter izkorišča pravilo, da zaviti oklepaji niso obvezni, če telo vsebuje en sam stavek. Na sliki 5 je prikazan diagram poteka za zgornji primer stavka *if*. Graf vsebuje štiri vejitve (po dve za vsak stavek *if*) ter tri poti čez graf od začetne do končne točke.



Slika 5: Diagram poteka za stavek *if - else if - else*

Pri analizi stavka *if* s pomočjo postavljene bombe v vsako od vej *then* in *else* ugotovimo, ali se veji dejansko izvedeta in na osnovi tega zabeležimo število pokritih vejitev ter število vseh

vejitev (pri stavku *if* je to 2) za obravnavani stavek. Če stavek *if* nima veje *else*, jo moramo dodati v drevo AST:

```
if (ifStmt.getElseStmt() == null) {
    ifStmt.setElseStmt(new BlockStmt());
}
```

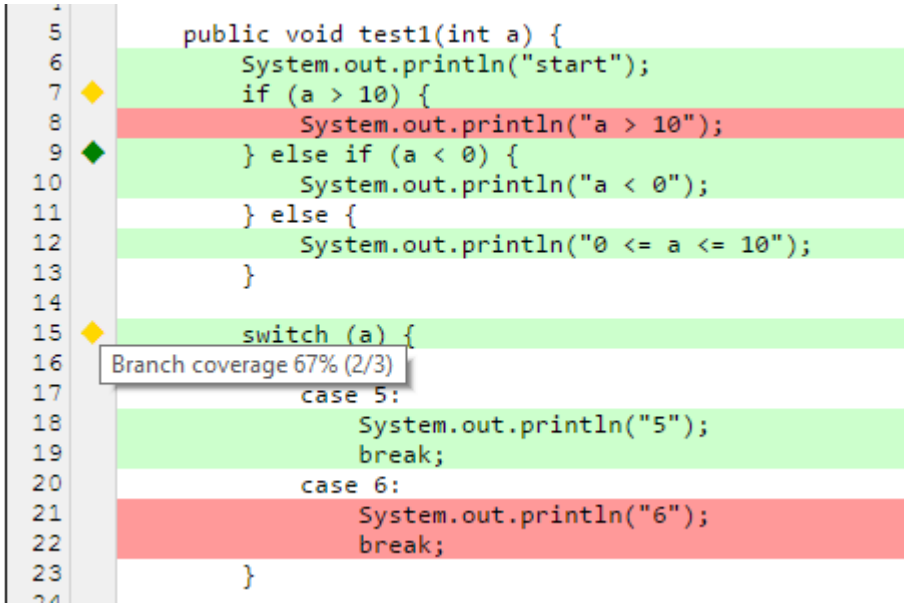
Na koncu še zabeležimo ugotovljeno pokritost vejitev za obravnavani stavek glede na vrstico izvorne kode, kjer se stavek nahaja:

```
javaFileData.setBranchCoverage(ifStmt.getBeginLine(),
        new BranchCoverage(nCoveredBranches, 2));
```

Pri stavku *switch* na podoben način s pomočjo postavljene bombe v vsako od vej ugotovimo, koliko vej se dejansko izvede. Ena od možnih vej je tudi veja *default*, ki ni obvezna, zato jo moramo v primeru, če manjka, dodati programsko v drevo AST:

```
SwitchEntryStmt defaultEntryStmt = new SwitchEntryStmt(null, new ArrayList<>());
switchStmt.getEntries().add(defaultEntryStmt);
```

Pokritost vejitev grafično prikažemo v poročilu s pomočjo simbola romb v posebnem stolpcu na levi strani v vsaki vrstici, ki vsebuje vejitev. Primer prikaza je na sliki 6.



Slika 6: Grafični prikaz pokritosti vejitev

Barva simbola pove stopnjo pokritosti vejitev:

- zelena barva pomeni polno (100 %) pokritost

- rumena barva pomeni delno pokritost
- rdeča barva pomeni nično (0 %) pokritost

4.7 Prevajanje izvorne kode

Za prevajanje mutirane izvorne kode je zadolžen razred *InMemoryCompiler*. Osnovno Javansko orodje za prevajanje je orodje ukazne vrstice *javac*, ki kot vhod sprejme javansko datoteko na datotečnem sistemu. Temu odvečnemu koraku shranjevanja mutirane izvorne kode na datotečni sistem smo se želeli izogniti (s čimer bi tudi prepisali originalno izvorno datoteko v Maven projektu), kar nam je uspelo z razredom javax.tools.JavaCompiler in razširitvijo razreda *SimpleJavaFileObject*.

Konstruktor razreda *InMemoryCompiler* zahteva dva parametra: *outputDirectory* in *compileClasspath*. Parameter *outputDirectory* predstavlja pot do mape, kamor se shranjujejo prevedene (*class*) datoteke, parameter *compileClasspath* pa pot do razredov z vsemi knjižnicami, potrebnimi med prevajanjem:

```
InMemoryCompiler inMemoryCompiler =
        new InMemoryCompiler(outputDirectory, compileClasspath);
```

Razred *InMemoryCompiler* ponuja metodo *compile*, ki prevede podano javansko izvorno kodo razreda podanega s polno kvalificiranim imenom *classFQN*:

```
void compile(String sourceCode, String classFQN)
```

Za prevajanje ustvarimo instanco razreda *JavaCompiler*, pripravimo vse potrebne parametre in kreiramo objekt *CompilationTask*:

```
JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
DiagnosticCollector<JavaFileObject> diagnostics = new
        DiagnosticCollector<JavaFileObject>();
List<String> compileOptions = new ArrayList<String>();
compileOptions.addAll(Arrays.asList("-d", outputDirectory.getAbsolutePath()));
compileOptions.addAll(Arrays.asList("-classpath", compileClasspath));
JavaFileObject file = new JavaSourceFromString(classFQN, sourceCode);
Iterable<? extends JavaFileObject> compilationUnits = Arrays.asList(file);
CompilationTask task = compiler.getTask(null, null, diagnostics, compileOptions,
        null, compilationUnits);
```

Razred *JavaSourceFromString* razširja razred *SimpleJavaFileObject* in omogoča prevajanje izvorne kode, shranjene v pomnilniku kot objekt tipa *String*.

Prevajanje sprožimo z metodo *call*:

```
boolean success = task.call();
```

Vrednost spremenljivke *success*, ki jo vrne metoda, nam pove, ali je bilo prevajanje uspešno. Če je pri prevajanju prišlo do napake, lahko iz objekta *DiagnosticCollector* dobimo podroben opis napake.

Prevedena datoteka (s končnico *class*) se shrani na datotečni sistem v podano mapo *outputDirectory* (*target/classes* v standardni Maven strukturi map). Vedno se prevaja celotna datoteka z javansko izvorno kodo. Če ta vsebuje več razredov ali gnezdene razrede, ne moremo prevesti samo izbranega, ampak vedno datoteko kot celoto.

4.8 Izvajanje testnih enot

Ko v izbrano javansko datoteko postavimo bombo in jo prevedemo, moramo izvesti testne enote in preveriti, ali je bila vstavljena bomba odkrita. Izvajanje testov smo implementirali na dva načina. Najprej s pomočjo knjižnice JUnit, nato še bolj splošno s pomočjo vtičnika Maven Surefire[2]. Pri zagonu orodja BBJCC lahko s parametrom *testRunner* izberemo, katerega od obeh načinov želimo uporabiti.

4.8.1 S pomočjo knjižnice JUnit

Razred *JUnitCore*, del knjižnice JUnit, omogoča programsko poganjanje testnih enot *JUnit* s pomočjo naslednje metode:

```
Result run(java.lang.Class<?>... classes)
```

Podati moramo seznam vseh razredov s testnimi enotami, katere želimo pognati. Seznam razredov dobimo tako, da preiščemo mapo s testno izvorno kodo, kar smo že opisali v poglavju 4.3. Preden lahko poženemo teste, moramo rešiti še dve težavi:

- Pri prevajanju mutiranih javanskih razredov se prevedena koda zapiše v *class* datoteke na datotečnem sistemu, v pomnilniku pa so še vedno stare verzije razredov.
- Kontekstni nalagalnik razredov (angl. classloader) v vtičniku Maven ima dostop samo do odvisnosti vtičnika (definirane v datoteki *pom.xml* vtičnika), ne pa do razredov in odvisnosti projekta, ki ga analiziramo, zato ne more naložiti razredov, potrebnih za izvajanje testnih enot (dobimo napako *ClassNotFoundException*).

[2] http://maven.apache.org/surefire/maven-surefire-plugin/

Prvo težavo smo rešili s pomočjo orodja Hotswap Agent [9], ki omogoča zamenjavo oziroma ponovno naložitev (angl. hot-swap) že naloženega javanskega razreda v pomnilniku. V osnovi orodje spremlja *class* datoteke na disku, zazna spremembo in ponovno naloži spremenjene razrede. Za naše potrebe smo ta sistem nekoliko spremenili, ker se ta akcija izvrši z nekoliko zamika in ne vrne nobene povratne informacije, kdaj je končana. Logiko smo zapakirali v metodo *hotswap*, ki kot parameter sprejme polno kvalificirano ime razreda, poišče ustrezno *class* datoteko na datotečnem sistemu in razred ponovno naloži:

```
void hotswap(String classFQN)
```

Drugo težavo smo rešili s kreiranjem nove instance nalagalnika razredov, kateremu podamo pot do razredov analiziranega projekta kot seznam objektov *URL*:

```
ClassLoader unitTestClassloader = URLClassLoader.newInstance(testClasspathUrls,
        Thread.currentThread().getContextClassLoader());
```

Seznam *testClasspathUrls* vsebuje:

- pot do mape s prevedenimi razredi analiziranega projekta (mapa *target/classes*)
- pot do mape s prevedenimi testnimi razredi analiziranega projekta (mapa *target/test-classes*)
- vse odvisnosti analiziranega projekta z dometom *test*

S pomočjo novega nalagalnika razredov naložimo testne razrede in pripravimo njihov seznam:

```
List<Class> unitTestClasses = new ArrayList<Class>();
for (String unitTestClassName : unitTestClassNames) {
    Class unitTestClass = unitTestClassloader.loadClass(unitTestClassName);
    unitTestClasses.add(unitTestClass);
}
```

Seznam testnih razredov pretvorimo v polje, kreiramo instanco *JUnitCore* in izvedemo teste:

```
JUnitCore junit = new JUnitCore();
Result result = junit.run(unitTestClassesArray);
```

Če so se testi uspešno izvedli, to pomeni, da bomba ni bila odkrita in tisti del kode torej ni pokrit. Če so testi oziroma vsaj eden od testov spodleteli, moramo še preveriti, če je vzrok za napako res vstavljena bomba. Iz objekta *Result* dobimo seznam napak, iz napake pa izjemo s sporočilom. Sporočilo izjeme se mora ujemati s sporočilom, ki smo ga zapisali v bombo (stavek *assert*), to je »*mutation_i*«.

4.8.2 S pomočjo vtičnika Maven Surefire

Drugi, bolj splošen način za izvajanje testov je s pomočjo vtičnika za orodje Maven, imenovanim Maven Surefire[3]. To je standardni vtičnik za orodje Maven, ki se uporablja med testno fazo gradnje projekta za izvajanje testnih enot. Pri običajni uporabi se vtičnik požene iz ukazne vrstice z ukazom *mvn test*, v našem primeru pa ga moramo pognati programsko iz aplikacije, ki je prav tako vtičnik za Maven. Za to nalogo smo uporabili orodje Mojo Executor[4], ki je tudi vtičnik za Maven. Teste poženemo s klicem metode *executeMojo*, kot parametre podamo podatke o vtičniku in cilj, ki ga želimo zagnati:

```
executeMojo(
        plugin(
                groupId("org.apache.maven.plugins"),
                artifactId("maven-surefire-plugin"),
                version("2.19.1")
        ),
        goal("test"),
        configuration(
                element(name("argLine"), "-D" + UUID.randomUUID().toString())
        ),
        executionEnvironment(
                mavenProject,
                mavenSession,
                pluginManager
        )
);
```

Vtičnik Surefire si zapomni kontekst, v katerem je bil pognan in se ne zažene dvakrat zaporedoma v isti fazi gradnje projekta z istimi parametri. To omejitev smo zaobšli s pomočjo naključnega parametra *UUID*, ki ga dodamo kot parameter ukazne vrstice pri zagonu vtičnika.

Za razliko od prvega načina se v tem primeru testi izvajajo v svojem procesu. Vsakič, ko poženemo vtičnik Surefire, se ustvari nov proces, v katerem se izvedejo testi. Zaradi tega je ta način bolj robusten v primerjavi s prvim.

4.9 Izdelava poročila

Ko zaključimo z analizo izvorne kode, moramo izdelati še zaključno poročilo. Poročilo zapišemo v mapo *target/site/bbjcc*, v skladu s standardno strukturo map orodja Maven. Poročilo je izdelano v formatu *HTML*, pri obliki poročila smo se zgledovali po orodju EMMA.

[3] http://maven.apache.org/surefire/maven-surefire-plugin/
[4] https://github.com/TimMoore/mojo-executor

Surovi podatki, zbrani med analizo izvorne kode, vsebujejo za vsako javansko datoteko naslednje podatke:

- pot do datoteke
- ime paketa
- polje vrednosti *LineStatus[]*, ki vsebuje podatek o pokritosti za vsako vrstico datoteke. *LineStatus* je naštevni tip z vrednostmi *COVERED*, NOT_*COVERED* in *UNDEFINED*.
- polje vrednosti *BranchCoverage[]*, ki vsebuje podatke o pokritosti vejitev za posamezno vrstico oziroma *null*, če v vrstici ni vejitev. *BranchCoverage* je razred z atributoma *nCovered* (število pokritih vejitev) in *nBranches* (število vseh vejitev).
- logična vrednost, ali datoteka vsebuje izvršljivo kodo
- seznam javanskih razredov v datoteki ter za vsak razred naslednje podatke:
    - ime razreda
    - začetna in končna vrstica razreda
    - število metod ter koliko od teh metod je pokritih

Na osnovi teh podatkov izdelamo poročilo, ki vsebuje naslednje podatke:

- na nivoju celotnega projekta: pokritost (število pokritih / število vseh) in delež pokritosti za razrede, metode in vrstice izvorne kode, statistika projekta (število vseh paketov, datotek z izvršljivo kodo, razredov, metod in vrstic z izvršljivo kodo), seznam paketov.
- na nivoju paketa: pokritost (število pokritih / število vseh) in delež pokritosti za razrede, metode in vrstice izvorne kode, seznam datotek.
- na nivoju datoteke: pokritost (število pokritih / število vseh) in delež pokritosti za razrede, metode, vrstice izvorne kode in vejitve, seznam vsebovanih razredov ter pokritost po posameznih razredih, izpisek izvorne kode in grafični prikaz pokritosti kode in vejitev.

Poročilo je izdelano v obliki datotek *HTML* - za vsak paket, javansko datoteko in celoten projekt po ena datoteka. Za generiranje poročila smo uporabili javansko knjižnico Apache FreeMarker [10], ki omogoča delo s predlogami. Pripravili smo predlogo za vsak tip poročila (projekt, paket,

datoteka). To so datoteke v formatu *FTL* (angl. FreeMarker Template Language), ki vsebujejo kodo *HTML* z direktivami za FreeMarker. Ko ta procesira predlogo, izvede direktive in namesto njih vstavi ustrezno vsebino.

Knjižnico FreeMarker uporabljamo na naslednji način. Najprej ustvarimo instanco razreda *Configuration* ter podamo pot do mape, kjer so shranjene predloge:

```
Configuration cfg = new Configuration(Configuration.VERSION_2_3_23);
cfg.setClassForTemplateLoading(this.getClass(), "/templates");
cfg.setDefaultEncoding("UTF-8");
```

S pomočjo objekta *cfg* ustvarimo objekt *Template* za izbrano predlogo:

```
Template template = cfg.getTemplate("package-report.ftl");
```

Pripravimo slovar parametrov z ustreznimi podatki za izbrano predlogo:

```
Map<String, Object> input = new HashMap<String, Object>();
input.put("packageName", packageName);
input.put("packageSummary", packageSummary);
…
```

V predlogi parameter *packageSummary* vstavimo s pomočjo direktive ${}:

```
<td>${packageSummary.lineCoverage}</td>
```

Poženemo procesiranje predloge, izhod zapišemo v datoteko:

```
Writer fileWriter = new FileWriter(new File(reportFilesDirectory,
                    reportFileName));
template.process(input, fileWriter);
```

Na slikah 7, 8, 9 in 10 je prikazan primer poročila, generiranega za projekt Jsontoken[5].

[5] https://github.com/google/jsontoken

BBJCC Coverage Report (generated 24.07.2016 18:22:22)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	line, %
all classes	92% (23/25)	83% (105/127)	63% (334/526)

OVERALL STATS SUMMARY

```
total packages:          4
total executable files:  25
total classes:           25
total methods:           127
total executable lines:  526
```

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	line, %
net.oauth.jsontoken	100% (4/4)	100% (4/4)	86% (176/204)
net.oauth.jsontoken.crypto	90% (9/10)	90% (9/10)	57% (68/119)
net.oauth.jsontoken.discovery	80% (4/5)	80% (4/5)	28% (15/54)
net.oauth.signatures	100% (6/6)	100% (6/6)	50% (75/149)

[all classes]

BBJCC 1.0 (C) Damjan Murn

Slika 7: Povzetek poročila za celoten projekt

BBJCC Coverage Report (generated 24.07.2016 18:22:22)

[all classes]

COVERAGE SUMMARY FOR PACKAGE [net.oauth.jsontoken]

name	class, %	method, %	line, %
net.oauth.jsontoken	100% (4/4)	100% (4/4)	86% (176/204)

COVERAGE BREAKDOWN BY SOURCE FILE

name	class, %	method, %	line, %
Checker.java	N/A	N/A	N/A
Clock.java	N/A	N/A	N/A
JsonToken.java	100% (1/1)	96% (26/27)	85% (95/112)
JsonTokenParser.java	100% (1/1)	82% (9/11)	88% (67/76)
JsonTokenUtil.java	100% (1/1)	71% (5/7)	71% (5/7)
SystemClock.java	100% (1/1)	100% (4/4)	100% (9/9)

[all classes]

BBJCC 1.0 (C) Damjan Murn

Slika 8: Poročilo za izbrani paket

BBJCC Coverage Report (generated 24.07.2016 18:22:23)

[all classes][net.oauth.jsontoken]

COVERAGE SUMMARY FOR SOURCE FILE [JsonToken.java]

name	class, %	method, %	line, %
JsonToken.java	100% (1/1)	96% (26/27)	85% (95/112)

COVERAGE BREAKDOWN BY CLASS AND METHOD

name	method, %	line, %	branch, %
JsonToken	96% (26/27)	85% (95/112)	81% (21/26)

```
1  /**
2   * Copyright 2010 Google Inc.
3   *
4   * Licensed under the Apache License, Version 2.0 (the "License");
5   * you may not use this file except in compliance with the License.
6   * You may obtain a copy of the License at
```

Slika 9: Poročilo za izbrano javansko datoteko

```
241   public SignatureAlgorithm getSignatureAlgorithm() {
242     if (sigAlg == null) {
243       if (header == null) {
244         throw new IllegalStateException("JWT has no algorithm or header");
245       }
246       JsonElement algorithmName = header.get(JsonToken.ALGORITHM_HEADER);
247       if (algorithmName == null) {
248         throw new IllegalStateException("JWT header is missing the required '" +
249             JsonToken.ALGORITHM_HEADER + "' parameter");
250       }
251       sigAlg = SignatureAlgorithm.getFromJsonName(algorithmName.getAsString());
252     }
253     return sigAlg;
254   }
255
256   public String getTokenString() {
257     return tokenString;
258   }
259
260   public JsonObject getHeader() {
261     if (header == null) {
262       createHeader();
263     }
264     return header;
265   }
```

Slika 10: Grafični prikaz pokritosti kode in pokritosti vejitev

Poglavje 5 Uporaba

Orodje BBJCC je namenjeno za analizo pokritosti kode za javanske projekte, upravljane z orodjem Maven. Za delovanje potrebuje naslednje:

- javanski razvojni komplet (JDK 8)
- Apache Maven 3

Izvorna koda je na voljo na spletnem servisu GitHub. Projekt prenesemo s pomočjo orodja *git*:

```
git clone https://bitbucket.org/damjanmurn/bbjcc.git
```

Premaknemo se v korensko mapo projekta BBJCC ter ga z naslednjim ukazom zgradimo in namestimo v lokalni repozitorij Maven:

```
mvn install
```

Premaknemo se v projekt, ki ga želimo analizirati, ga prevedemo in izvedemo testne enote:

```
mvn clean test
```

Orodje BBJCC poženemo kot vtičnik za Maven z ukazom:

```
mvn damjan.bbjcc:bbjcc-maven-plugin:analyse
```

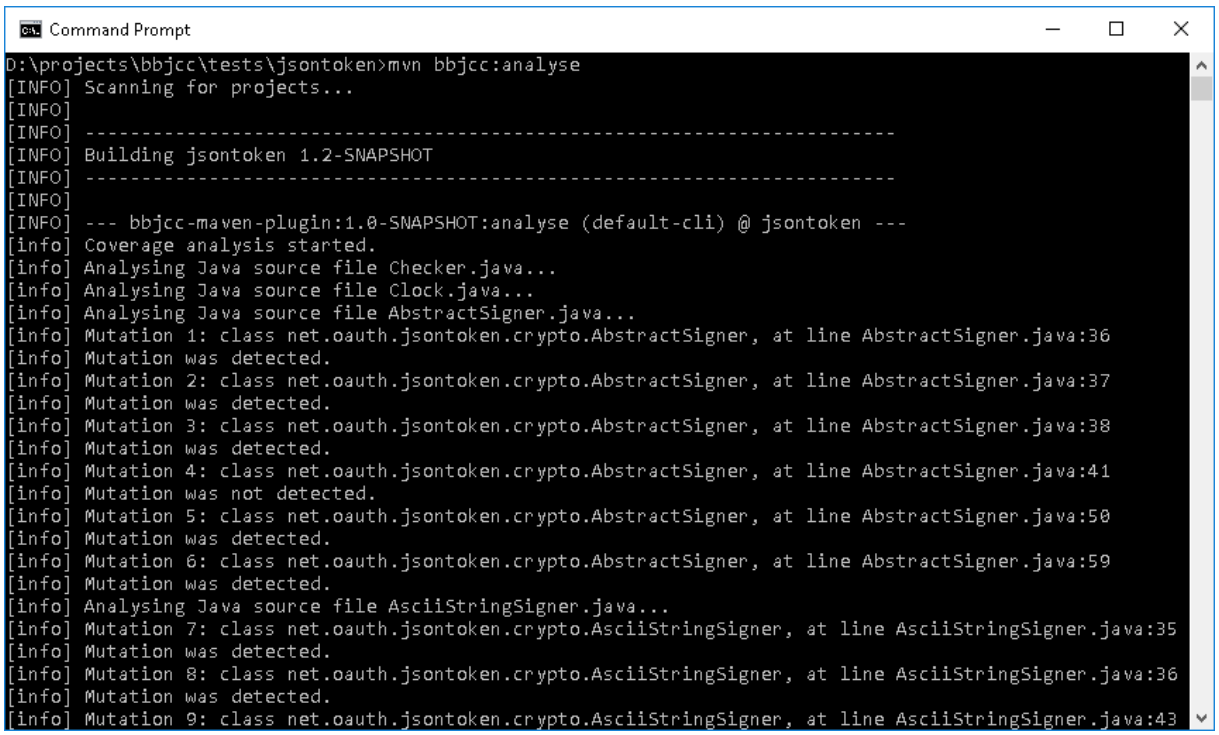
Ukaz lahko skrajšamo, če dodamo atribut *groupId* vtičnika v seznam *pluginGroups* v nastavitveni datoteki orodja Maven *settings.xml*, ki se nahaja v mapi *.m2* v uporabnikovi domači mapi:

```
<pluginGroups>
  <pluginGroup>damjan.bbjcc</pluginGroup>
</pluginGroups>
```

V tem primeru lahko analizo poženemo z ukazom:

```
mvn bbjcc:analyse
```

Slika 11 prikazuje zagon analize iz ukazne vrstice za projekt Jsontoken[6].



Slika 11: Zagon orodja BBJCC iz ukazne vrstice

Ko se analiza zaključi, lahko poročilo najdemo v mapi *target/site/bbjcc* znotraj obravnavanega projekta.

5.1 Primerjava rezultatov

Za testiranje orodja BBJCC ter primerjavo in umerjanje z orodjem EMMA smo med razvojem uporabljali testni projekt *bbjcc-test-samples* s primeri vseh konstruktov programskega jezika Java v različnih oblikah. Na koncu smo si izbrali še tri ne preveč obsežne, prosto dostopne javanske projekte in pognali analizo pokritosti kode. Na istih projektih smo za primerjavo analizo izvedli še z orodjem EMMA. Zraven smo zabeležili še čas izvajanja, pri čemer je analiza tekla na računalniku s procesorjem Intel Core i7-6700 s 4 jedri in s frekvenco delovanja 3.4 GHz. Rezultati so zbrani v tabelah 2, 3, 4 in 5.

	BBJCC	**EMMA**
Število paketov, datotek, razredov	1, 12, 16	1, 12, 16

[6] https://github.com/google/jsontoken

Pokritost razredov	88% (14/16)	88% (14/16)
Pokritost metod	74% (51/69)	74% (51/69)
Pokritost vrstic	61% (211/344)	62% (238,6/382)
Čas izvajanja	5 min 28 s	2 s

Tabela 2: Primerjava rezultatov za projekt bbjcc-test-samples

	BBJCC	**EMMA**
Število paketov, datotek, razredov	4, 25, 25	4, 25, 25
Pokritost razredov	92% (23/25)	92% (23/25)
Pokritost metod	84% (112/134)	82% (111/135)
Pokritost vrstic	63% (334/526)	67% (373,4/561)
Čas izvajanja	7 min 28 s	2 s

Tabela 3: Primerjava rezultatov za projekt Jsontoken[7]

	BBJCC	**EMMA**
Število paketov, datotek, razredov	3, 31, 40	3, 31, 40
Pokritost razredov	90% (36/40)	75% (30/40)
Pokritost metod	52% (185/356)	50% (179/358)
Pokritost vrstic	55% (415/757)	55% (471,6/860)
Čas izvajanja	11 min 10 s	4 s

Tabela 4: Primerjava rezultatov za projekt Google OAuth Java Client[8]

	BBJCC	**EMMA**
Število paketov, datotek, razredov	6, 49, 150	6, 49, 159
Pokritost razredov	95% (142/150)	96% (152/159)
Pokritost metod	80% (631/792)	85% (746/882)
Pokritost vrstic	85% (3255/3833)	85% (3669/4317)
Čas izvajanja	1 h 36 min	12 s

[7] https://github.com/google/jsontoken
[8] https://github.com/google/google-oauth-java-client

Tabela 5: Primerjava rezultatov za projekt Gson[9]

Vidimo lahko, da se rezultati precej dobro ujemajo. Pri pokritosti razredov in metod je nekaj odstopanja, pri čemer razlike nastanejo v podrobnostih. EMMA npr. šteje k metodam tudi statični inicializacijski blok, katerega dejansko sploh ni v izvorni kodi, če ima razred kakšno statično lastnost objektnega tipa z inicializacijo. Poleg tega se k metodam štejejo tudi implicitni konstruktor, implicitni metodi *valueOf* in *values* pri naštevnem podatkovnem tipu, implicitni konstruktor pri anonimnih razredih… Zaradi tega je težko vse take posebnosti obravnavati na enak način kot EMMA.

Pokritost vrstic v odstotkih se zelo lepo ujema, v absolutnih številkah pa so določene razlike (EMMA ima v vseh primerih višje številke). Razlike nastanejo v podrobnostih, kot so npr. označevanje zaključnih zavitih oklepajev, označevanje deklaracije metod in konstruktorjev, označevanje lastnosti razreda. Poleg tega EMMA ni vedno dosledna, v nekaterih primerih npr. označi (šteje kot izvršljivo) vrstico z deklaracijo razreda, običajno ne, pri metodah ne označuje deklaracije metode, pri konstruktorjih pa v nekaterih primer ja, v drugih ne.

Popolnega ujemanja dejansko ni mogoče doseči, ker EMMA deluje na nivoju vmesne kode in pokritost, ugotovljeno na osnovnih blokih vmesne kode, projicira na vrstice javanske izvorne kode. Orodje BBJCC pa deluje na nivoju javanske izvorne kode.

Ogromna razlika pa nastane pri času izvajanja analize. EMMA je zelo učinkovita in analizo opravi v nekaj sekundah, BBJCC pa je bistveno počasnejši, za projekt velikosti cca 4000 izvršljivih vrstic potrebuje uro in pol, pri čemer se ta razlika z velikostjo projekta še povečuje. Mutacijska analiza je že v osnovi zelo počasna in računsko zahtevna. EMMA deluje na nivoju vmesne kode, na začetku izvede instrumentalizacijo kode, tako da vanjo vstavi posebne zastavice, in izvede teste, ki se izvedejo le enkrat. BBJCC pa deluje po principu mutacijskega testiranja na nivoju javanske izvorne kode, pri čemer mora prevajati javansko kodo in poganjati teste za vsako mutacijo posebej. Ravno izvajanje testov je časovno najbolj potratna operacija.

Sliki 12 in 13 prikazujeta izsek grafičnega prikaza pokritosti kode iz poročil, izdelanih z orodjem BBJCC ter z orodjem EMMA, za isti del izvorne kode. Rumena barva v poročilu orodja EMMA pomeni delno pokritost vrstice, torej da se je tista vrstica izvorne kode samo delno izvršila (npr. v logičnem izrazu se je izvršil le prvi operand). Tega BBJCC ne preverja, vendar pa v nasprotju z EMMO prikaže še pokritost vejitev z oznakami v obliki romba različnih barv v stolpcu na levi strani.

[9] https://github.com/google/gson

```
284     protected String computeSignatureBaseString() {
285       if (baseString != null && !baseString.isEmpty()) {
286         return baseString;
287       }
288       baseString = JsonTokenUtil.toDotFormat(
289           JsonTokenUtil.toBase64(getHeader()),
290           JsonTokenUtil.toBase64(payload)
291           );
292       return baseString;
293     }
294
295     private JsonObject createHeader() {
296       header = new JsonObject();
297       header.addProperty(ALGORITHM_HEADER, getSignatureAlgorithm().getNameForJson());
298       String keyId = getKeyId();
299       if (keyId != null) {
300         header.addProperty(KEY_ID_HEADER, keyId);
301       }
302       return header;
303     }
304
305     private String getSignature() throws SignatureException {
306       if (signature != null && !signature.isEmpty()) {
307         return signature;
308       }
309
310       if (signer == null) {
311         throw new SignatureException("can't sign JsonToken with signer.");
312       }
313       String signature;
314       // now, generate the signature
315       AsciiStringSigner asciiSigner = new AsciiStringSigner(signer);
316       signature = Base64.encodeBase64URLSafeString(asciiSigner.sign(baseString));
317
318       return signature;
319     }
320
321   }
```

Slika 12: Izsek iz poročila, izdelanega z orodjem BBJCC

```
284   protected String computeSignatureBaseString() {
285     if (baseString != null && !baseString.isEmpty()) {
286       return baseString;
287     }
288     baseString = JsonTokenUtil.toDotFormat(
289         JsonTokenUtil.toBase64(getHeader()),
290         JsonTokenUtil.toBase64(payload)
291         );
292     return baseString;
293   }
294
295   private JsonObject createHeader() {
296     header = new JsonObject();
297     header.addProperty(ALGORITHM_HEADER, getSignatureAlgorithm().getNameForJson());
298     String keyId = getKeyId();
299     if (keyId != null) {
300       header.addProperty(KEY_ID_HEADER, keyId);
301     }
302     return header;
303   }
304
305   private String getSignature() throws SignatureException {
306     if (signature != null && !signature.isEmpty()) {
307       return signature;
308     }
309
310     if (signer == null) {
311       throw new SignatureException("can't sign JsonToken with signer.");
312     }
313     String signature;
314     // now, generate the signature
315     AsciiStringSigner asciiSigner = new AsciiStringSigner(signer);
316     signature = Base64.encodeBase64URLSafeString(asciiSigner.sign(baseString));
317
318     return signature;
319   }
320
321 }
```

Slika 13: Izsek iz poročila, izdelanega z orodjem EMMA

Poglavje 6 Sklepne ugotovitve

V diplomskem delu smo na principu mutacijskega testiranja z uporabo mutacijskega operatorja BSR razvili orodje za analizo pokritosti programske kode s testi, ki smo ga poimenovali BBJCC. Pristop se je pokazal kot uporaben, z dobrimi možnostmi za raziskovanje delovanja programske kode med izvajanjem testov. Pri testiranju orodja na realnih projektih se je pokazalo, da so rezultati dokaj natančni in primerljivi z orodjem EMMA. Popolnega ujemanja pa ne bi bilo mogoče doseči, ker EMMA deluje na nivoju javanske vmesne kode in ne na nivoju javanske izvorne kode kot BBJCC. Za EMMA-o je osnovna enota pri analizi pokritosti t.i. osnovni blok, zaporedje ukazov vmesne kode brez skokov. Za BBJCC pa je osnovna enota javanski stavek.

Kot enega od ciljev smo si na začetku postavili, da orodje BBJCC po funkcionalnosti in načinu merjenja pokritosti čim bolj posnema EMMA-o. Podprli smo večino funkcionalnosti in metrik orodja EMMA, z izjemo metrike pokritost osnovnih blokov ter detekcije delne pokritosti vrstic. Metrika pokritost osnovnih blokov je pogojena z analizo na nivoju vmesne kode in je zato ni bilo mogoče podpreti. Detekcija delne pokritosti vrstic bi zahtevala podrobno analizo logičnih izrazov in smo jo pustili kot eno od možnih izboljšav. Namesto tega pa smo implementirali analizo pokritosti vejitev, česar EMMA ne podpira. BBJCC pri pogojnih stavkih preveri, ali so se izvedle vse vejitve in to tudi grafično in številčno prikaže v končnem poročilu.

Največji problem orodja BBJCC je počasnost in neučinkovitost, kar je povezano s samim principom delovanja. Algoritem smo optimizirali, kolikor se je dalo, vendar je že v osnovi princip delovanja zelo neučinkovit. Po vsaki mutaciji oziroma vstavljeni bombi je potrebno javansko izvorno kodo prevesti. Ta proces smo optimizirali, tako da se prevajanje izvrši v pomnilniku in ni potrebno zapisovati izvorne kode na disk. Poleg tega se prevede samo spremenjena javanska datoteka in ne celoten projekt. Če algoritem ugotovi, da nek del kode oziroma vozlišče v drevesu AST ni pokrit, preneha s postavljanjem bomb in samo označi podrejena vozlišča kot nepokrita.

Najbolj časovno potratno pa je izvajanje zbirke testov po vsaki mutaciji, še posebej pri večjih projektih z veliko testi. Število mutacij se približno ujema s številom vseh izvršljiv stavkov v projektu, ki so pokriti s testi oziroma se dejansko izvedejo med izvajanjem testov. Časovna zahtevnost algoritma je torej sorazmerna s produktom števila pokritih vrstic in časa izvajanja

zbirke testov. Če približno ocenimo, da čas izvajanja zbirke testov linearno narašča z velikostjo projekta, je torej časovna zahtevnost algoritma kvadratna funkcija velikosti projekta.

Za konec še nekaj idej, kako bi aplikacijo lahko izboljšali in nadgradili z novimi funkcionalnostmi. S pomočjo postavljanja bomb v logične izraze bi lahko preverjali *pokritost pogojev* (angl. *condition coverage*, tudi *predicate coverage*). To je metrika, ki ugotavlja, ali je vsak operand vsakega logičnega izraza v programu zavzel obe možni logični vrednosti, to je *resnično* in *neresnično* [5]. Mehanizem za beleženje pokritosti bi lahko nadgradili tako, da bi se pokritost namesto na nivoju vrstic beležila bolj podrobno in bi lahko v poročilu označili vrstice, ki so samo delno pokrite. Izvajanje testov po vsaki mutaciji, najbolj časovno zahtevno operacijo, bi se splačalo optimizirati tako, da bi se izvajanje prekinilo takoj ko spodleti prvi testni primer. Orodje, ki smo ga uporabili za izvajanje testov, omogoča samo izvajanje testne zbirke kot celote, pri čemer se izvedejo vsi testni primeri, na koncu pa dobimo rezultate po posameznih testnih primerih.

Literatura

[1] P. Ammann, J. A. Offutt, *Introduction to Software Testing*, Cambridge University Press, New York, NY, ZDA, 2008.

[2] Y. Jia, M. Harman, *An Analysis and Survey of the Development of Mutation Testing*, IEEE Transactions on Software Engineering. Dostopno na: http://www0.cs.ucl.ac.uk/staff/mharman/tse-mutation-survey.pdf.

[3] V. Okun, *Specification Mutation for Test Generation and Analysis*, PhD thesis, University of Maryland Baltimore. Dostopno na: http://csrc.nist.gov/groups/SNS/acts/documents/thesis_vadim.pdf.

[4] Techopedia, *Mutation testing*, Techopedia Inc. Dostopno na: https://www.techopedia.com/definition/20905/mutation-testing.

[5] Wikipedia, *Code coverage*, Wikimedia Foundation, Inc. Dostopno na: https://en.wikipedia.org/wiki/Code_coverage.

[6] EMMA: a free Java code coverage tool. Dostopno na: http://emma.sourceforge.net/.

[7] Apache Maven. Dostopno na: https://maven.apache.org/

[8] Java Parser and Abstract Syntax Tree. Dostopno na: http://javaparser.org/.

[9] Hotswap Agent. Dostopno na: http://www.hotswapagent.org/.

[10] Apache FreeMarker. Dostopno na: http://freemarker.org/.